View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Tennessee, Knoxville: Trace

University of Tennessee, Knoxville
Trace: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

5-2002

Using the Maple Computer Algebra System as a Tool for Studying Group Theory

Thomas Edmond Cooper, III
University of Tennessee - Knoxville

Recommended Citation

Cooper, III, Thomas Edmond, "Using the Maple Computer Algebra System as a Tool for Studying Group Theory. " Master's Thesis, University of Tennessee, 2002.
https://trace.tennessee.edu/utk_gradthes/2042

This Thesis is brought to you for free and open access by the Graduate School at Trace: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of Trace: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Thomas Edmond Cooper, III entitled "Using the Maple Computer Algebra System as a Tool for Studying Group Theory." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mathematics.

Lawrence Husch, Jr, Major Professor

We have read this thesis and recommend its acceptance:

Ken Stephenson, Charles Collins

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council

I am submitting herewith a thesis written by Thomas Edmond Cooper, III entitled "Using the Maple Computer Algebra System as a Tool for Studying Group Theory." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mathematics.

Lawrence Husch, Jr
Major Professor

We have read this thesis and
recommend its acceptance:

Ken Stephenson

Charles Collins

Acceptance for the Council:

Dr. Anne Mayhew
Vice Provost and Dean of
Graduate Studies

(Original signatures are on file in the Graduate Student Services Office.)

USING THE MAPLE COMPUTER ALGEBRA SYSTEM AS A TOOL FOR STUDYING GROUP THEORY

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Thomas Edmond Cooper, III

May 2002

Copyrighted © 2001 by Thomas E. Cooper, III
All rights reserved.

DEDICATION

This dissertation is dedicated to my wife, Mary Owens Cooper, for all of her love and support throughout the process of preparing this thesis and throughout my entire graduate career. It is also dedicated to my parents, Tom Cooper, Jr. and Ann Cooper, for supporting me and giving me the opportunities to become what I am today. This thesis is also dedicated to my father-in-law and mother-in-law, Steve Owens and Mary Owens, for all of their support.

ACKNOWLEDGEMENTS

I wish to thank all those who helped me in completing my degrees at the University of Tennessee, Knoxville. I thank Dr. Husch for serving as my major professor and for all of his help in completing this thesis from beginning to end. I thank Dr. Stephenson and Dr. Collins for serving on my committee, and I thank all three of my committee members for being outstanding teachers.

ABSTRACT

The purpose of this study was to show that computers can be powerful tools for studying group theory. Specifically the author examined ways that the computer algebra system Maple can be used to assist in the study of group theory. The study consists of four main parts.

After a brief introduction in chapter one, chapter two discusses simple procedures written by the author to study small finite groups. These procedures rely on the fact that for small finite groups, the elements can all be stored on a computer and tested for various properties. All of the procedures are contained in the appendix, and each is described in chapter two.

The Maple software comes with a built in set of group theory procedures. The procedures work with two types of groups, permutation groups and finitely presented groups. The author discusses all of the procedures dealing with permutation groups in chapter three and the procedures for finitely presented groups in chapter four. The main theoretical tool for permutation groups is a stabilizer chain, and the main tool for finitely presented groups is the Todd-Coxeter algorithm. Both of these methods and their implementations in Maple are discussed in detail.

The study is concluded by examining some applications of group theory. The author discusses check digit schemes, RSA encryption, and permutation factoring. The ability to factor a permutation in terms of a set of generators can be used to solve several puzzles such as the Rubik's cube.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Over the past few decades, computers have become very important tools for mathematicians. From calculating logarithms and trigonometric values to solving complex differential equations, computers can now, in mere seconds, do many of the tedious computations that once required hours of hand calculations. Few people would argue that it is to one's advantage to utilize such powerful tools.

It is obvious that computers are excellent at performing numerical calculations. From differential equations to linear programming, computers have been widely used to crunch numbers for large iterative algorithms. It may not be as obvious how a computer can be used to study an abstract branch of mathematics such as modern algebra. One's first impressions might be that there is no way that a machine can deal with abstract entities such as groups and rings. However, with a little human ingenuity, computers can do amazing things with sets and operations. While computers cannot compare to the human mind, they can be used to perform some tasks that would otherwise be too time consuming for practicality. Computers also allow one to examine large amounts of information quickly and efficiently. It is for these reasons that computers can play an important role in the study of algebra.

The content of this thesis will focus on using a computer to study group theory. Many other divisions of abstract algebra have made use of computers, such as ring theory and Galois Theory. A complete account of computing in abstract algebra could easily fill many volumes of work. Computational group theory, or studying groups with the aid of computers, has grown tremendously over the last few decades. In fact this specialized area has grown too large to cover completely in one volume. Thus this paper will focus primarily on the study of groups using the software package Maple.

It is the intent of this paper to show that computers can be utilized to study abstract algebra by working with algorithms written by the author for small finite groups, exploring Maple's group package in depth, and discussing a few possible applications. There are numerous software packages and programming languages available that could be used for this purpose. The author has chosen to use Maple, the symbolic algebra system designed by the Symbolic Computation Group at the University of Waterloo in Ontario, Canada.

There are several reasons that Maple was selected over other software for this report. First of all, Maple is widely available to the mathematics community. Most university mathematics departments have access to Maple, and individuals can purchase it for private use at a reasonable price. Secondly, Maple comes complete with a group package that implements some of the leading algorithms in computational group theory. Furthermore, Maple allows the user to examine the codes used in the group package. The command interface(verboseproc=2) changes the interface allowing one to examine internal routines in detail. Once the interface has been changed, eval(command) will print the code that Maple uses for a particular command. The author will not reproduce directly any of Maple's algorithms, but the reader is strongly encouraged to look at them using the Maple software.

Maple was also selected because of its use in other fields of mathematics. If one is going to invest a serious amount of time in learning to use a software package, it is good to use one such as Maple that can be utilized for various branches of mathematics. The software GAP, which can freely be obtained on the Internet, is specifically written to perform computational group theory. GAP is a great deal more advanced at group theory than Maple, but Maple can be used for a wider range of mathematics.

This report will consist of three main parts. First of all, the author will discuss procedures that he has written for working with small finite groups. Then Maple's group package will be discussed. This package consists of two primary parts, algorithms for permutation groups and algorithms for groups defined by generators and relations. A tremendous amount of theory has been developed for handling groups of these two types. Maple takes advantage of this material, and the author will discuss it in detail. Finally, the report will conclude with some applications of group theory. There are many applications of group theory. The author will discuss permutation puzzles, error correcting, and RSA encryption.

This thesis should convince most readers that computers can and should play an important role in all branches of mathematics. The algorithms presented are not extremely complicated. A strong undergraduate math course would probably cover the amount of theory necessary to understand these procedures. As with all branches of mathematics, some of the simplest results involve a great deal of ingenuity. The algorithms presented in this thesis provide the initial step to combine conventional mathematics with modern technology. How far mathematicians can go with this technology is limited only by the human mind.

CHAPTER 2: ALGORITHMS FOR SMALL FINITE GROUPS

INTRODUCTION

This chapter will focus on simple algorithms written by the author to work with small finite groups. The procedures are based on ideas found in Dubinsky and Leron's book on the program ISETL [DL]. These procedures are in the file FGv1.mpl and listed in the appendix. In theory, the procedures work on any finite group, but in practice there is a limit on storage space and computation time. These procedures will serve two main purposes. First of all they should enrich one's understanding of groups, allowing one to examine several small finite groups quickly on a computer. Many people learn things visually; therefore, seeing the groups and subgroups on a computer screen can reinforce the knowledge obtained from books for many people. Secondly, these procedures introduce some basic ideas of representing groups on a computer. The next two chapters will focus on more advanced techniques for studying much larger groups. The algorithms in this chapter can serve to familiarize one with Maple and the group properties necessary to perform the more advanced algorithms.

Throughout this report, it is assumed that the reader has some experience with group theory and the Maple software. Some useful references are listed in the bibliography. For background information on groups, [Ga1] is a good starting point, and [CG] is a good reference for Maple commands. Whenever necessary, certain key facts and definitions will be presented, such as the following definition.

Definition 2.1: A set G together with an operation $\circ$ form a **group** if together they satisfy the following:

a) $a \circ b \in G$ for all $a,b \in G$.

b) $a \circ (b \circ c) = (a \circ b) \circ c$ for all $a,b,c \in G$

c) There exists an element $e \in G$ such that $e \circ a = a \circ e = a$ for all $a \in G$.

d) For each $a \in G$, there exists an element $a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = e$

Frequently when describing an abstract group, one denotes $a \circ b$ as ab.

BASIC GROUP PROPERTIES

If a set G is finite and the operation can be defined on a computer, then the properties of definition 2.1 can be tested on a computer by examining every single element in the set. That is what some of the procedures in the author's file FGv1.mpl do. There are procedures Closed, Assoc, Identity, and Inverses that test a-d respectively.

Closure

The procedure Closed can be called with two or three arguments. The optional third argument determines how much information is returned. Once a set G and an operation $\circ$ have been defined, the set

can be tested for closure under the operation. The command is either Closed(G, ◦) or Closed(G, ◦, x) with anything entered as the third argument. If only two arguments are used, then the output is *closed* or *not closed*. If the optional third argument is entered, then the procedure will not only return *not closed*, if that is the case, but it will also return two elements a and b in G such that a◦b is not in G. The procedure iterates through all possible combinations of elements a and b in the given set. If at any point two element a and b are found such that a◦b is not in the set, then G is not closed under ◦, and the appropriate response is returned. If all possible combinations of elements are exhausted and a◦b∈ G for all of them, then the set is closed under ◦, and the appropriate response is returned.

It is well known that the set of integers {0, 1, ..., n-1} is a group under addition modulo n for any positive integer n. The usual notation for these groups is Z_n. These groups can easily be represented in Maple and will therefore serve as a primary example. The set can be stored with the command Zn := [seq(i-1, i = 1..n)] with the appropriate n. The operation can be stored using the command f:= (a, b) ->(a+b) mod n. Once f is defined in this way, f(a, b) will return the value of (a+b) mod n. FGv1.mpl contains a procedure Zmodn that will set up Z_n. Once the file has been read into Maple, the command Zmodn(n, f) sets up Zn = [0, 1, 2, ..., n-1] with the operation f = (a, b) → (a+b) mod n.

This gives an example to use with Closed. If one uses the command Zmodn(6, f), then Z_6 is set up, and typing Closed(Zn, f) returns *closed*. The command Closed(Zn, f, x) returns the message *This set with this operation is closed*. One should expect Zn to be closed under f. What happens if f is changed to f:= (a, b) → (a+b)? Then Closed(Zn, f) returns *not closed*, and Closed(Zn, f, x) returns the message *this set with this operation is not closed since f(1, 5)=6.*

Associativity

The procedure Assoc is used to test if a set G with an operation f is associative, i.e., satisfies property b of definition 2.1. The procedure Assoc uses the same set of arguments as Closed. As with Closed, a third argument causes the procedure to return additional information. The algorithm searches through all possible combinations of elements a, b, and c in G and tests to see if f(a, f(b, c))=f(f(a, b), c). If one set of elements a, b, and c is found that does not satisfy the property, then the set is not associative under f. Otherwise, it is an associative set under f. If Z_6 has been defined with Zmodn, then Assoc(Zn, f) returns *associative*. The command Assoc(Zn, f, x) returns the message *this set with this operation is associative.* Defining f:= (a, b) → a+b would return similar results since addition is associative. Suppose g:=(a, b) → (a-b) mod 6. Then Assoc(Zn, g) returns *not associative*, and Assoc(Zn, g, x) returns *this set with this operation is not associative since (0 * 0)* 1 = 5 and 0 * (0 * 1)= 1.*

A set G together with an operation f that is closed and associative is called a semigroup. A semigroup that has an identity, i.e., an element that satisfies property c of definition 2.1, is called a monoid, and a monoid with inverses is a group. The procedures Closed and Assoc can be used to determine whether or not something is a semigroup. The procedures Identity and Inverses will be necessary to test if something is a group.

Identity Elements

The procedure Identity searches through the elements of a set G one at a time looking for an identity. To test if an element x is the identity, xg and gx are calculated for all g in G. If one g is found such that gx or xg is not equal to g, then that element x is not the identity, and the algorithm jumps to the next element. If an element x is found such that gx = xg = g for all g in G, then it is returned as the identity. If no such x is found, *noid* is returned, meaning there is no identity.

Using Identity with Z_6 by typing Identity(Zn, f) returns *0.* Clearly 0 is the identity of Z_6 since (0 mod 6) + (x mod 6) = (0+x) mod 6 = x mod 6 for all $x \in \mathbf{Z}$. Using g = (a, b)→(a-b) mod n, Identity(Zn, g) returns *noid*. There is no optional third argument for Identity, but there is another procedure Isidentity in FGv1.mpl. This procedure determines whether or not a specific element is the identity of a set under an operation. For instance, Isidentity(0, Zn, f) returns the message *0 is the identity of this set*. The command Isidentity(1, Zn, f) returns the message *1 is not the identity since 1*0 = 1*. Hence the procedure not only determines whether or not an element is the identity. It provides proof that certain elements are not the identity.

Inverse Elements

The procedure Inverse tries to find the inverse of a particular element in a given set under a given operation. The command Inverse(a, G, f) searches the elements of G looking for one such that f(g, a) = f(a, g) = e_G. Hence the identity of G must first be defined. Using the example Z_6, Inverse(1, Zn, f) returns *5*. The command Inverse(1, Zn, f, x) returns the message *1 inverse is 5 since 1 * 5 = 5 * 1 = 0*. Suppose h:=(a, b) → (ab) mod 6. Then Inverse(2, Zn, h) returns *noinv* since 2 has no inverse under multiplication modulo 6. The command Inverse(2, Zn, h, x) returns the message *2 has no inverse in this set*.

The procedure Inverses determines whether or not every element in the set has an inverse by using the command Inverse(a, G, f) for all $a \in G$. Since Z_6 is a group, Inverses(Zn, f) returns *inverses*. The command Inverses(Zn, f, x) returns the message *Every element in this set has an inverse*. Using g:= (a, b)→(a-b) mod 6 and h:= (a, b)→(ab) mod 6, Inverses(Zn, g) and Inverses(Zn, h) return *no identity implies no inverses* and *no inverses* respectively. The command Inverses(Zn, h, x) returns the message *0 has no inverse*.

Testing Groups with Isgp

Using the commands Closed, Assoc, Identity, and Inverses, the procedure Isgp can test a finite set with an operation to determine if they form a group. The procedure tests all four of the properties in definition 2.1. If f:= (a, b)→(a+b) mod 6, g:= (a, b)→(a-b) mod 6, h:= (a, b)→(ab) mod 6, and Zn:= [0, 1, 2, 3, 4, 5], then the commands Isgp(Zn, f), Isgp(Zn, g), and Isgp(Zn, h) return *Group*, *not a group*, and *not a group* respectively. Adding an optional third argument allows one to see what happens with each property. The command Isgp(Zn, f, x) returns the following:

The operation is closed and associative.

The set has 0 and inverses

Hence this set with this operation is semigrp, monoid, and grp

The command Isgp(Zn, g, x) returns the following:

The operation is closed and `not associative`.

The set has noid and `no identity implies no inverses`

Hence this set with this operation is notsemigp, notmonoid, and notgrp

The command Isgp(Zn, h, x) returns the following:

The operation is closed and associative.

The set has 1 and `no inverses`

Hence this set with this operation is semigrp, monoid, and notgrp

It should be apparent that these procedures in the file FGv1.mpl allow one to quickly test a set and operation to see how they behave under the properties of definition 2.1. It is the author's belief that these procedures can strongly reinforce one's understanding of the basic properties of a group. When a certain property fails for a set and operation, the procedures can show the user why they fail, not just that they do fail.

Commutativity

Two elements a and b in a group G are said to commute if $a \circ b = b \circ a$. Definition 2.1 does not require that $a \circ b = b \circ a$ for all a, $b \in G$. A group for which $a \circ b = b \circ a$ for all $a,b \in G$ is called a commutative group or an Abelian group in honor of Norwegian mathematician Neils Abel. The procedure Comm tests all possible combinations of a and b in G for commutativity. If a pair is found that does not commute, then the group is not Abelian. If all pairs commute, then the group is Abelian. If Zmodn has been run for some n, then the command Comm(Zn, f) returns *commutative*. The command Comm(Zn, f, x) returns *this set with this operation is commutative*. The command can be used for any set and operation even if they do not form a group. For instance if Zn:= [0, 1, 2, 3, 4, 5], g:=(a, b)→(a-b) mod 6, and h:=(a, b)→(ab) mod 6, then Comm(Zn, g, x) returns *this set with this operation is not commutative since 0 * 1 = 5 and 1 * 0 = 1,* and Comm(Zn, h, x) returns *this set with this operation is commutative.*

THE GROUPS IN FGV1.MPL

The procedures in this chapter work with any set and operation that one can define on a computer. There are many ways to define sets and operations. The file FGv1.mpl contains procedures to load many familiar finite groups. There are procedures for Z_n, U(n), S_n, the quaternions, permutation groups, direct sums, $T_{12} = \langle a, b \mid a^6 = b^{-2}a^3 = bab^{-1}a = 1 \rangle$, and other groups defined by generators and relations. Some

groups such as Z_n are easily represented in Maple by defining sets and functions as previously shown. Some non-Abelian groups such as T_{12} can be defined using a multiplication table to define the operation. This section will discuss the groups in FGv1.mpl one at a time.

The Integers Modulo n

The easiest groups to set up on a computer are the integers under addition modulo n. The set is just a list [0, 1, 2, ..., n-1], and the operation is addition modulo n which most programming languages, including Maple, can perform. The command Zmodn(n, f) sets up the list Zn:= [0, 1, 2, ..., n-1] and the operation f:= (a, b) →(a+b) mod n for any positive integer n. For example, typing Zmodn(12, f) returns the following on the Maple worksheet:

Zn=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
(a, b) -> (a + b) mod 12

The size of the groups that can be created by this procedure is limited only by storage space and the user's patience. The author has run the procedure to produce up to $Z_{1,000,000}$.

The Units under Multiplication Modulo n

The set of integers [0, 1 ,..., n-1] does not form a group under multiplication modulo n since 0 has no multiplicative inverse modulo n, but for each positive integer n the set U(n) ={x∈ {0, 1, 2, ..., n-1}| gcd(x, n) =1} forms a group under multiplication modulo n . The command Umodn(n, f) forms U(n) with f:=(a, b)→(ab) mod n for any positive integer n. The procedure stores all integers x between 0 and n-1 such that gcd(x, n) =1 into a list named Un. For example, typing Umodn(12, f) would return the following:

Un = [1, 5, 7, 11]
(a, b) -> a b mod 12

The procedure Isgp can verify that Un is indeed a group with operation f. Isgp(Un, f, x) returns:

The operation is closed and associative.
The set has 1 and inverses
Hence this set with this operation is semigrp, monoid, and grp

Permutation Groups

The symmetric group of degree n, S_n, and other permutation groups can be set up using Maple's group package. This package will be discussed in great detail in the next two chapters. Maple does not have a command to list or store the elements of a permutation group, but it does have procedures that can be utilized. Maple's notation for a permutation expressed as a product of disjoint cycles is to represent each cycle in brackets and enclose them all in brackets. For example the cycle (1, 2, 3) is [[1, 2, 3]], and (1, 2)(3, 4) is [[1, 2], [3, 4]]. The command permgroup simply sets up the degree and generators for a

permutation group that can be used with other procedures. One of these procedures is cosets, which returns a complete set of coset representatives for a subgroup. This can be used to list all of the elements in a group. The group can be defined with generators, such as S4:= permgroup(4, {[[1, 2, 3, 4]], [[1, 2]]}). Then the trivial subgroup sg := permgroup(4, {[]}) can be formed. The procedure cosets(sg, S4) will list all of the coset representatives for sg in S4, which are precisely the elements of S4. Maple has a procedure mulperms for finding the composition of a permutation. These two things are used to define a permutation group with the procedure pgp. For example the command pgp(4, {[[1, 2, 3, 4]], [[2, 4]]) stores the permutation group of degree 4 generated by the set {[[1, 2, 3, 4]], [[2, 4]]} into a list called Pgp. This creates the set Pgp:= [[], [[2, 4]], [[1, 2], [3, 4]], [[1, 2, 3, 4]], [[1, 3]], [[1, 3], [2, 4]], [[1, 4, 3, 2]], [[1, 4], [2, 3]]], which is D_4, the dihedral group of order 8.

The entire symmetric group of degree n can be defined using Maple's combinatorics package. The procedure permute(n) returns the complete list of permutations of the integers 1 to n. These are presented as lists of length n, where the i^{th} entry is the image of the integer i under the permutation. The command convert can be used to convert these permutation lists to products of disjoint cycles. The procedure Sn uses this process to construct the symmetric group of degree n.

There is another way to list the elements of a permutation group without using Maple's built in package. This is to use Dimino's algorithm [Bu p.20], named after its inventor Lou Dimino. A simple but inefficient way to enumerate the elements of a group is to take advantage of closure. If a and b are elements of a group G, then $ab \in G$. Hence, one can find all of the elements of a group by multiplying together all combinations of elements already known, starting with the generators, until no new elements are found. A great deal of the products will be redundant. Since every element is a product of the generators, it is only necessary to multiply each known element by each generator. This is still very inefficient.

Dimino's idea is to take advantage of cosets. Since a group G is the disjoint union of its cosets [Hu p.38], one can enumerate elements of a group cosets at a time. Suppose that a subgroup H of G has been enumerated. Now suppose that an element g is found such that g is not in H. Then the entire coset Hg can be added, i.e., xg is a unique element of G for each x in H. This is the key fact that Dimino's algorithm uses to enumerate the elements of a group. A simple version of Dimino's algorithm is found in [Bu p.20]. The author has included a Maple adaptation of the algorithm in FGv1.mpl. The command Dimino(L) can be used to generate a permutation group with a list of permutations L. For example Dimino([[[1, 2, 3, 4]], [[1, 2]]]) will list the elements of S_4.

The Quaternions and T_{12}

The quaternions and T_{12} can be defined using multiplication tables. The sets can be stored as Q8:= [e, a, a^2, a^3, b, ba, ba^2, ba^3] and T12:= [1, A, B, C, D, E, F, G, H, I, J, K]. The operations can be defined by storing complete multiplication tables as matrices or arrays. The multiplication tables for the quaternions and T_{12} are in figure 1. The function defined on these tables finds ab by finding the locations i and j of a and b in the set and letting ab = the [i, j] entry of the table. For example, BJ = TT[3, 11] = F. The

(a) The quaternions

$$\begin{bmatrix} e & a & a^2 & a^3 & b & ba & ba^2 & ba^3 \\ a & a^2 & a^3 & e & ba^3 & b & ba & ba^2 \\ a^2 & a^3 & e & a & ba^2 & ba^3 & b & ba \\ a^3 & e & a & a^2 & ba & ba^2 & ba^3 & b \\ b & ba & ba^2 & ba^3 & a^2 & a^3 & e & a \\ ba & ba^2 & ba^3 & b & a & a^2 & a^3 & e \\ ba^2 & ba^3 & b & ba & e & a & a^2 & a^3 \\ ba^3 & b & ba & ba^2 & a^3 & e & a & a^2 \end{bmatrix}$$

(b) T_{12}

$$\begin{bmatrix} 1 & A & B & C & D & E & F & G & H & I & J & K \\ A & B & C & D & E & 1 & G & H & I & J & K & F \\ B & C & D & E & 1 & A & H & I & J & K & F & G \\ C & D & E & 1 & A & B & I & J & K & F & G & H \\ D & E & 1 & A & B & C & J & K & F & G & H & I \\ E & 1 & A & B & C & D & K & F & G & H & I & J \\ F & K & J & I & D & G & C & B & A & 1 & E & D \\ G & F & K & J & G & H & D & C & B & A & 1 & E \\ H & G & F & K & H & I & E & D & C & B & A & 1 \\ I & H & G & F & I & J & 1 & E & D & C & B & A \\ J & I & H & G & J & K & A & 1 & E & D & C & B \\ K & J & I & H & K & F & B & A & 1 & E & D & C \end{bmatrix}$$

Figure 1. The multiplication tables for the quaternions and T_{12}

commands q8( ) and t12( ) store the sets Q8 and T12 and the arrays QT and TT. The functions are defined in FGv1.mpl as qmult and tmult.

Finitely Presented Groups

A common way to define a group is with a set of generators and relations [Ga1 p. 438]. Maple has procedures for dealing with these types of groups, but there is no procedure to list the elements of a group and no procedure to perform multiplication. This can be taken care of with a little work. Just like the permutation groups, cosets can be used to list group elements. A group such as T_{12} can be defined with the command G:= grelgroup({a, b}, {[a, a, a, a, a, a], [1/b, 1/b, a, a, a], [b, a, 1/b, a] }). Then the trivial subgroup can be defined as H:= subgrel({z=[a, a, a, a, a, a]}, G). Finally, the elements of G can be listed with the command cosets(H). Unlike permutation groups, multiplication is not easy to define. It can however be done with coset representatives. For a word x in the generators of G, the command cosrep(x, H) returns an element h of H = {1}, and a coset representative g such that hg = x. The internal procedure `group/mulword` forms the concatenation of two words. For example the command `group/mulword`([a, a, a],[b, a, b]) returns the word [a, a, a, b, a, b]. Such an element can then be simplified with cosrep(x, H). Since $h \in H$ implies that h = 1, $x = hg = g \in G$. These things are done in the routine Grelgroup in the file FGv1.mpl. Grelgroup({gens}, {relations}, f) sets up a group defined by the generators and relations with multiplication done by f. This provides an alternate way to form the quaternions and T_{12}.

Direct Sums

One way to form new groups from old ones is to form the direct sum of two groups. The procedure Dsum can be used to define direct sums of groups. The procedure takes n groups and sets up the set DS of all possible lists of length n where the i^{th} entry comes from the i^{th} group in the list of groups. Multiplication is then performed component wise by the function fds. Since all finite Abelian groups can be expressed as a direct sum of groups of integers under modular arithmetic [Ga1 p. 209], Dsum can theoretically be used to construct any finite Abelian group. For example, suppose that one wishes to construct $Z_2 \oplus Z_3 \oplus Z_2$. First, one can define the operations f:= (a, b)→(a+b) mod 2 and g:= (a, b)→(a+b) mod 3. Then the group can be defined by Dsum([[0, 1],[0, 1, 2],[0, 1]], [f, g, f]). The command returns the message *DS and fds loaded*. Then DS:= [[0, 0, 0], [1, 0, 0], [0, 1, 0], [1, 1, 0], [0, 2, 0], [1, 2, 0], [0, 0, 1], [1, 0, 1], [0, 1, 1], [1, 1, 1], [0, 2, 1], [1, 2, 1]]. Isgp(DS, fds, x) returns the following:

The operation is closed and associative.
The set has [0, 0, 0] and inverses
Hence this set with this operation is semigrp, monoid, and grp

In theory, all of these procedures give access to any finite group as long as a finite presentation can be found. However, computers have a limited amount of storage space, and users have a limited amount of time. The procedures in this chapter are primarily intended for studying small finite groups.

MISCELLANEOUS PROCEDURES IN FGV1.MPL

The file FGv1.mpl contains several procedures for working with groups once they have been defined. There are procedures Ctable, newf, elpow, elord ,elords, sbgp, subgps, subgpsord, cyca, iscyclic, Center, Centralizer, Normalizer, Conjugate, Lcoset, Rcoset, HK, Lcosets, isnorm, GmodH, and fntabel. In this section, these procedures will be discussed one by one.

Cayley Tables and Functions Defined By Them

One tool for studying groups is a multiplication table also known as a Cayley table in honor of mathematician Arthur Cayley [Ga1 p. 31]. Cayley tables for the quaternions and T_{12} have already been shown. The procedures q8 and t12 use Cayley tables to define the group operations. The procedure Ctable forms the Cayley table for a group and stores it as a global variable CT. For example, if Z_6 has been defined using Zmodn(6, f), then Ctable(Zn, f) will form the Cayley table for Z_6 and return the message *CT is loaded*. The procedure does not display the Cayley table since these tables can become quite large. The table can be examined by typing print(CT). For this example, the Cayley table is as follows:

$$\text{CT} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 0 \\ 2 & 3 & 4 & 5 & 0 & 1 \\ 3 & 4 & 5 & 0 & 1 & 2 \\ 4 & 5 & 0 & 1 & 2 & 3 \\ 5 & 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

The procedure newf defines a function using the Cayley table formed by Ctable. The procedure assumes that the first row and first column of CT correspond to the group. Then multiplication is defined as with any multiplication table.

Powers of Elements

Given a group G with operation f, the procedure elpow computes a^n for any $a \in G$ and any integer n. If n = 0, then the identity is returned. If n>0, then a loop calculates f(a, a, ..., a) where there are n copies of a. If n<0, then a loop calculates f(a^{-1}, a^{-1}, ..., a^{-1}) where there are n copies of a^{-1}. For example, suppose that q8 has been run. Then elpow(a, 7, Q8, qmult) returns a^3. The commands elpow(a, 0, Q8, qmult) and elpow(a, -29, Q8, qmult) return e and a^3 respectively.

Orders of Elements

The order of an element g in a group G with identity e is the smallest positive integer n such that $g^n = e$. The procedure elord finds the order of an element by calculating powers of that element until the

identity is obtained. The procedure elords returns a list containing the orders for all of the elements of a group. Suppose that T_{12} has been defined using t12( ). Then elord(D, T12, tmult) returns *3*, and elords(T12, tmult) returns the list *[1, 6, 3, 2, 3, 6, 4, 4, 4, 4, 4, 4]*.

Subgroups

Definition 2.2: A subset of a group G that is a group itself under the operation of G is called a **subgroup** of G.

Theorem 2.1: If H is a nonempty finite subset of a group, then H is a subgroup of G if it is closed under the operation of G.

Proof: See Gallian [Ga1 p. 60].

The procedure sbgp uses theorem 2.1 to test if a subset of a group forms a subgroup by checking closure. Suppose that Z_4 has been set up using Zmodn(4, f). Then sbgp([0, 2], Zn, f) returns *Yes*. The commands sbgp([0, 1], Zn, f) and sbgp([0, 1, 2, 3, 4], Zn, f) return *false*. The set [0, 1] fails closure, and [0, 1, 2, 3, 4] is not a subset of [0, 1, 2, 3].

Finding Subgroups of a Given Order

Theorem 2.2 (Lagrange's Theorem): Let G be a group and H a subgroup of G. Then the order of H divides the order of G.

Proof: See Gallian [Ga1 p. 135-136].

The procedure subgpsord searches a group G for all of the subgroups of order n for a positive integer n. First the procedure uses Lagrange's theorem to determine whether or not it is possible to obtain a subgroup of order n. If it is not, then the procedure returns a message stating this fact. If n = 1, then the identity is returned. If n is equal to the order of G, then G is returned. Otherwise, the procedure uses the choose procedure from the combinatorics package to find all possible combinations of subsets of G with n elements. Any set not containing the identity is discarded. Then the remaining sets are tested using theorem 2.1. As the order of G grows, the number of subsets obtained by the command choose can be quite large. Hence, this procedure can be time consuming and is not practical for large groups.

Finding All Subgroups

The procedure subgps uses Lagrange's theorem to search for all of the subgroups of a group. It uses the choose command, as in subgpsord, for each of the divisors of the order of G. This procedure is

terribly time consuming for large groups, but it can be quite informative for small groups such as T_{12}. The procedure shows that the only subgroups of T_{12} are [1], [C, 1], [B, D, 1], [H, K, C, 1], [J, C, G, 1], [F, C, 1, I], [C, E, B, A, D, 1], and [H, K, J, F, C, E, B, A, G, D, 1, I] = T_{12}.

Cyclic Subgroups

Theorem 2.3: For any a in a group G, the set $\langle a \rangle = \{a^i \mid i \in \mathbf{N}\}$ forms a subgroup called the cyclic subgroup generated by a.

Proof: See Gallian [Ga1 p. 60].

The procedure cyca forms the cyclic subgroup of a group G generated by an element a. For example in the quaternions, cyca(a, Q8, qmult) returns the subgroup [a, a^2, a^3, e]. The procedure simply stores a^i into a list for i from 1 until a^i = e.

Testing if Groups are Cyclic

A group G is called cyclic if $G = \langle g \rangle$ for some $g \in G$. The procedure iscyclic tests to see if a group G is cyclic. Note that by theorem 2.3, $\langle g \rangle$ is a subgroup of G for all $g \in G$. It follows that $G = \langle g \rangle$ if and only if the order of g is equal to the order of G. The procedure iscyclic calls the procedure elords to calculate the order of each element in G. If an element is found such that |g| = |G|, then $G = \langle g \rangle$ and is cyclic. Otherwise, G is not cyclic. For example, if Zn:=[0, 1, 2, 3, 4] and f:=(a, b)→(a+b) mod 5, then iscyclic(Zn, f) returns *this group is cyclic generated by 1*. If sn(3) has been run, then iscyclic(Sn, mulperms) returns *not cyclic*.

The Center of a Group

Let $Z(G) = \{g \in G \mid gx = xg \text{ for all } x \in G\}$. This set is called the center of G, and it is easily shown that Z(G) is a subgroup of G [Ga1 p. 62]. The command Center(G, f) returns the center of a group G with operation f. The procedure simply tests each element of G to determine whether or not it commutes with all of the elements of G. For instance, Center(Q8, qmult) and Center(T12, tmult) show that $Z(Q8) = \{e, a^2\}$ and $Z(T_{12}) = \{1, C\}$.

The Centralizer of a Set

A subgroup similar to the center of a group is the centralizer of a set of elements. Let H be a subset of a group G. Then the centralizer of H in G, denoted $C_G(H)$, is the set $\{g \in G \mid gh = hg \text{ for all } h \in H\}$. If H is a subgroup of a group G, then the set $C_G(H)$ is easily seen to be a subgroup of G [Ga1 p.64]. The algorithm for the routine Centralizer is similar to the algorithm for Center. Centralizer searches the elements of g for elements that commute with each element in H. In Q8, Centralizer([a], Q8, qmult) returns [e, a, a^2, a^3]. The command Centralizer([a, b], Q8, qmult) returns [e, a^2] which is equal to Z(Q8).

The Normalizer of a Subgroup

Theorem 2.4: Let H be a subgroup of G. Then the set $N_G(H) = \{x \in G \mid xHx^{-1}=H\}$ is a subgroup of G.

Proof: By the one-step subgroup test [Ga1 p. 58], a nonempty subset S of a group G is a subgroup of G if $ab^{-1} \in S$ for each a and b in S Let H be a subgroup of a group G. Let $N = N_G(H)$. Clearly the identity is an element of N. Hence, N is nonempty. Suppose that a and b are in N. Then $aHa^{-1} = bHb^{-1} = H$. We need to show that $(ab)H(ab)^{-1} = H$. Note that $(ab)H(ab)^{-1} = (ab)H(b^{-1}a^{-1}) = \{(ab)h(b^{-1}a^{-1}) \mid h \in H\} = \{a(bhb^{-1})a^{-1} \mid h \in H\} = \{axa^{-1} \mid x \in bHb^{-1}\} = a(bHb^{-1})a^{-1}$. Hence $(ab)H(ab)^{-1} = a(bHb^{-1})a^{-1} = aHa^{-1} = H$. ■

The procedure Normalizer finds the normalizer of a subset of a group. It does not check to ensure that the subset is a subgroup. Hence, the output might not be a subgroup. Suppose that sn(4) has been run setting up Sn:= S_4 and Pgp:= ⟨{[[1, 2, 3, 4]], [[2, 4]]}⟩. Then Normalizer(Pgp, Sn, mulperms) returns *[[], [[2, 4]], [[1, 2], [3, 4]], [[1, 2, 3, 4]], [[1, 3]], [[1, 3], [2, 4]], [[1, 4, 3, 2]], [[1, 4], [2, 3]]]* = Pgp.

The Conjugate of a Subgroup

Let H be a subgroup of a group G. Then for any $x \in G$, the conjugate of H in G with respect to x is the set $xHx^{-1}=\{xhx^{-1} \mid h \in H\}$. Clearly the identity is in xHx^{-1}. Hence, xHx^{-1} is nonempty. Note that $(xhx^{-1})^{-1} = xh^{-1}x^{-1}$ for any $h \in H$, and $(xax^{-1})(xb^{-1}x^{-1}) = xab^{-1}x^{-1} \in xHx^{-1}$ for all a and b in H. Thus xHx^{-1} is a subgroup of G by the one-step test [Ga1 p. 58]. The procedure Conjugate(x, H, G, f) forms the set xHx^{-1} by calculating xhx^{-1} for all h in H. For instance, Conjugate(b, [e, a, a^2, a^3], Q8, qmult) gives [e, a, a^3, a^2]. In this example, $xHx^{-1} = H$. This is not always the case.

Cosets

Let H be a subgroup of a group G. For any x in G, the set $xH = \{xh \mid h \in H\}$ is called the left coset of H in G with respect to x. The set $Hx = \{hx \mid h \in H\}$ is called the right coset of H in G with respect to x. The procedures Lcoset and Rcoset find these sets for a subgroup H and an element x. For example, consider the subgroup H:=[[], [[2, 4]], [[1, 2], [3, 4]], [[1, 2, 3, 4]], [[1, 3]], [[1, 3], [2, 4]], [[1, 4, 3, 2]], [[1, 4], [2, 3]]] of S_4. The command Rcoset([[3, 4]], H, Pgp, mulperms) returns the set {[[3, 4]], [[2, 3, 4]], [[1, 2]], [[1, 2, 4]], [[1, 3, 2]], [[1, 3, 2, 4]], [[1, 4, 3]], [[1, 4, 2, 3]]}. The command Lcoset([[3, 4]], H, Pgp, mulperms) returns the set {[[3, 4]], [[2, 4, 3]], [[1, 2]], [[1, 2, 3]], [[1, 3, 4]], [[1, 3, 2, 4]], [[1, 4, 2]], [[1, 4, 2, 3]]}. Hence xH does not necessarily equal Hx.

The command Lcosets(H, G, f) lists the left coset xH for each x in G. For example if S_3 has been defined as Pgp using the command pgp(3, {[[1, 2]], [[2, 3]]}) and H:=[[], [[1, 2]]], then the command Lcosets(H, Pgp, mulperms) displays the following:

{[], [[1, 2]]}
{[], [[1, 2]]}
{[[2, 3]], [[1, 2, 3]]}
{[[1, 3, 2]], [[1, 3]]}
{[[2, 3]], [[1, 2, 3]]}
{[[1, 3, 2]], [[1, 3]]}

Note that there are some redundancies since xH may equal yH for $x \neq y$ in G.

Testing Normality

A subgroup H of a group G such that xH = Hx for all $x \in G$ is called a normal subgroup. Equivalent definitions for a normal subgroup are $xHx^{-1} \subset H$ for all $x \in G$ and $xHx^{-1} = H$ for all $x \in G$. The procedure isnorm(H, G, f) determines whether or not a subgroup is normal by testing each x in G for $xHx^{-1} = H$. If $xHx^{-1} = H$ for all $x \in G$, then *normal* is returned. Otherwise, the procedure returns a case where $xHx^{-1} \neq H$ and the message *the subgroup is not normal*. For example, isnorm([[], [[2, 4]], [[1, 2], [3, 4]], [[1, 2, 3, 4]], [[1, 3]], [[1, 3], [2, 4]], [[1, 4, 3, 2]], [[1, 4], [2, 3]]], Pgp, mulperms) returns the following:

*[[3, 4]] * [[], [[2, 4]], [[1, 2], [3, 4]], [[1, 2, 3, 4]], [[1, 3]], [[1, 3], [2, 4]], [[1, 4, 3, 2]], [[1, 4], [2, 3]]] * [[3, 4]]^-1 = {[], [[2, 3]], [[1, 2], [3, 4]], [[1, 2, 4, 3]], [[1, 3, 4, 2]], [[1, 3], [2, 4]], [[1, 4]], [[1, 4], [2, 3]]}*

the subgroup is not normal

The Set HK

Let H and K be subgroups of a group G. A useful set similar to cosets is the set HK = {hk | $h \in H$ and $k \in K$}. Like cosets HK is not necessarily a subgroup of G. It can be shown that if G = HK and $H \cap K = \{e\}$, then $G \approx H \oplus K$ [Ga1 p. 183-184]. This is one reason that group theorist are interested in HK. The routine HK computes the set HK by calculating hk for all pairs of elements $h \in H$ and $k \in K$. Suppose that S_3 has been defined using Pgp, H:= {[], [[1, 2]]}, and K:= {[], [[1, 3]]}. Then S:= HK(H, K, Pgp, mulperms) returns the set S:= {[], [[1, 2]], [[1, 2, 3]], [[1, 3]]}. Note that sbgp(S, Pgp) returns *No* since that set is not closed under the operation of S_3. Note that one should not define a set created by HK as a variable named HK since a set and procedure should not have the same name.

Factor Groups

If H is a normal subgroup of a group G, then the set G/H = {xH | $x \in G$} forms a group under the operation (xH)(yH)=xyH [Ga1 p.173], called a factor group. The procedure GmodH forms the set of all distinct left cosets of H in G and sets up the operation g:= (a, b) →Lcoset(f(a[1], b[1]), H, G, f) where f is the operation of G. The procedure does not test to see if H is a normal subgroup, hence the set formed may

not form a group. Let H:= [e, a, a^2, a^3] be a subgroup of Q8. Testing isnorm(H, Q8, qmult) returns *normal*. GmodH(H, Q8, qmult, g) returns the list GMODH:=[{a^2, a, e, a^3}, {ba^2, ba^3, b, ba}] and the operation g. Using Isgp(GMODH, g, x) returns:

The operation is closed and associative.
The set has {a^2, a, e, a^3} and inverses
Hence this set with this operation is semigrp, monoid, and grp

As has already been shown, the subgroup H:= ⟨[[1, 2, 3, 4]], [[2, 4]]⟩ is not normal in S_4. If pgp is used to define S_4 as Pgp and ⟨[[1, 2, 3, 4]], [[2, 4]]⟩ is stored as H, then GmodH(H, Pgp, mulperms, k) returns GMODH:= [{[], [[1, 2, 3, 4]], [[2, 4]], [[1, 2], [3, 4]], [[1, 3]], [[1, 3], [2, 4]], [[1, 4, 3, 2]], [[1, 4], [2, 3]]}, {[[2, 3]], [[2, 3, 4]], [[1, 2, 4, 3]], [[1, 2, 4]], [[1, 3, 2]], [[1, 3, 4, 2]], [[1, 4, 3]], [[1, 4]]}, {[[3, 4]], [[2, 4, 3]], [[1, 2]], [[1, 2, 3]], [[1, 3, 4]], [[1, 3, 2, 4]], [[1, 4, 2]], [[1, 4, 2, 3]]}] with the operation k:= (a, b) → Lcoset(mulperms(a[1], b[1]), H, G, mulperms). Since H is not normal in G, one would not expect GMODH to be a group. Isgp(GMODH, k, x) returns the following:

The operation is closed and `not associative`.
The set has {[], [[1, 2, 3, 4]], [[2, 4]], [[1, 2], [3, 4]], [[1, 3]], [[1, 3], [2, 4]], [[1, 4, 3, 2]], [[1, 4], [2, 3]]} and Inverses
Hence this set with this operation is notsemigp, notmonoid, and notgrp

Finding Finite Abelian Groups

One of the most powerful results of elementary group theory is the Fundamental Theorem of Finite Abelian Groups. This important theorem allows one to easily classify up to isomorphism all finite Abelian groups of a given order. The theorem, as stated by Gallian, asserts, "Every finite Abelian group is a direct product of cyclic groups of prime-power order. Moreover, the factorization is unique except for rearrangement of factors [Ga1 p. 209]." This gives an algorithm for finding finite Abelian groups. As Gallian notes, since every cyclic group of order p^k is isomorphic to the group of integers under addition modulo p^k, any finite Abelian group is isomorphic to a direct product of the form

$$Z_{p_1^{n_1}} \oplus Z_{p_2^{n_2}} \oplus \ldots \oplus Z_{p_t^{n_t}}$$

where the p_i's are not necessarily distinct primes and each n_i is a positive integer [Ga1 p. 209].

Suppose that n is a positive integer with prime decomposition as follows:

$$n = p_1^{n_1} p_2^{n_2} \ldots p_s^{n_s}$$

where each p_i is a distinct prime and each n_i is a positive integer. Since $|H \oplus K| = |H||K|$ for groups H and K, a group of the form $G_1 \oplus G_2 \oplus \ldots \oplus G_s$ where $|G_i| = p_i^{n_i}$ has order n. Clearly $G_1 \oplus G_2 \oplus \ldots \oplus G_s$ is a

finite Abelian group if each G_i is a finite Abelian group. In fact, Gallian shows that all finite Abelian groups of order n must be isomorphic to a group of this form [Ga1 p. 214 -215]. By the Fundamental Theorem of Finite Abelian Groups, each G_i is isomorphic to a direct product of cyclic groups of prime-power order. Clearly such a direct product must be of the form

$$Z_{p_i}^{m_1} \oplus Z_{p_i}^{m_2} \oplus \ldots \oplus Z_{p_i}^{m_w}$$

where $m_1+m_2+ \ldots +m_w = n_i$. The set $\{m_1, m_2, \ldots, m_w\}$ is called a partition of n_i. Hence, the Fundamental Theorem assures that there is a unique finite Abelian group of a particular prime-power order up to isomorphism for each partition of the power, and these are the only finite Abelian groups of that order. Furthermore, finite Abelian groups of order n can be formed from direct products of the groups of prime-power order for the divisors of n.

Suppose that G is a finite Abelian group isomorphic to a direct product of the form

$$Z_{p_1}^{n_1} \oplus Z_{p_2}^{n_2} \oplus \ldots \oplus Z_{p_t}^{n_t} .$$

Then the prime-power order of each direct summand is called an elementary divisor of G. The above paragraph gives a recipe for calculating all of the possible combinations of elementary divisors. The procedure fntabel returns an array containing all of the possible combinations of elementary divisors. Thus it reveals all of the finite Abelian groups of order n for a positive integer n. For example fntabel(2^5) returns the array [[[2, 2, 2, 2, 2]], [[2, 2, 2, 4]], [[2, 4, 4]], [[2, 2, 8]], [[4, 8]], [[2, 16]], [[32]]]. Thus the only finite Abelian groups of order 32 up to isomorphism are

$$Z_2 \oplus Z_2 \oplus Z_2 \oplus Z_2 \oplus Z_2,$$
$$Z_2 \oplus Z_2 \oplus Z_2 \oplus Z_4,$$
$$Z_2 \oplus Z_4 \oplus Z_4,$$
$$Z_2 \oplus Z_2 \oplus Z_8,$$
$$Z_4 \oplus Z_8,$$
$$Z_2 \oplus Z_{16},$$
$$Z_{32}.$$

The algorithm works with arrays instead of lists and seems to be quite efficient for large integers. On the author's personal computer with a 1.7 Gigahertz processor, it takes only 14.030 seconds to enumerate the 11208 groups of order $2^{30}9$.

PROCEDURES FOR DEALING WITH HOMOMORPHISMS

Definition 2.3: Let G_1 and G_2 be two groups. A **homomorphism** ϕ from G_1 to G_2 is a mapping $\phi:G_1 \rightarrow G_2$ such that $\phi(a \circ b) = (\phi(a))*(\phi(b))$ where $\circ$ is the operation on G_1 and * is the operation on G_2.

Definition 2.4: Let G_1 and G_2 be two groups. Let $\phi:G_1 \rightarrow G_2$ be a homomorphism. If $\phi(a) = \phi(b)$ implies that $a = b$ for $a, b \in G_1$, then the mapping is one to one, and ϕ is called a **monomorphism**.

Definition 2.5: Let G_1 and G_2 be two groups. Let $\phi:G_1 \rightarrow G_2$ be a homomorphism. If for any $y \in G_2$ there exists an $x \in G_1$ such that $\phi(x) = y$, then the mapping is onto, and ϕ is called an **epimorphism**.

Definition 2.6: A homomorphism of groups that is both one to one and onto is called an **isomorphism**.

Homomorphisms

The procedure Homom determines whether or not a map between two sets is a homomorphism. Mappings can be defined in many ways using Maple. When looking at Z_n, an easy way to define a map from Z_n to Z_m is h:= (a) -> (a*k) mod m. Suppose that Z4:= [0, 1, 2, 3], Z6:= [0, 1, 2, 3, 4, 5], j:= (a, b) -> (a+b) mod 4, and g:= (a, b) → (a+b) mod 6. Define k1:=(a)→0, k2:=(a)→(a) mod 6, k3:=(a)→(2*a) mod 6, and k4:=(a)→(3*a) mod 6. Then Homom(k1, Z4, j, Z6, g) returns *homomorphism*. Homom(k2, Z4, j, Z6, g) returns:

*k2(1*3)=0 but k2(1)*k2(3)=4*

Does not preserve operations

Homom(k3, Z4, j, Z6, g) returns:

*k3(1*3)=0 but k3(1)*k3(3)=2*

Does not preserve operations

The command Homom(k4, Z4, j, Z6, g) returns *homomorphism*. It can be shown that the number of homomorphisms from Z_n to Z_m is equal to gcd(n, m). Hence k1 and k4 are the only homomorphism from Z_4 to Z_6.

The Kernel of a Homomorphism

Definition 2.7: Let $\phi:G \rightarrow H$ be a group homomorphism. Then the **kernel** of ϕ is defined to be the set $\ker \phi = \{ g \in G \mid \phi(g) = e_H \}$.

The procedure kernel finds the kernel of a map. If $f:G \rightarrow H$ is a group homomorphism, it can be shown that $f(e_G) = e_H$ [Ga1 p. 120] and that f is a monomorphism if and only if $\ker f = \{e_G\}$ [Hu p. 31]. It can also be shown that ker f is a normal subgroup of G [Ga1 p. 196]. Hence the kernel is very important in group theory. Suppose that k1 and k4 are defined as above. The commands kernel(k1, Z4, j, Z6, g) and kernel(k4, Z4, j, Z6, g) return *[0, 1, 2, 3]* and *[0, 2]* respectively. Note that both results are subgroups of

Z4, and they are normal since Z4 is Abelian. Since neither [0, 1, 2, 3] nor [0, 2] is equal to [0], k1 and k4 are not monomorphisms.

The Image of a Homomorphism

Definition 2.8: Let $f:G \rightarrow H$ be a group homomorphism. Let S be a subset of G. Then the **image** of S under f is the set $f(S) = \{ h \in H \mid h = f(x) \text{ for some } x \in S \}$.

The command image(f, S, op) finds the image of a subset S of a group G with operation op under a mapping f. Observe that if $f:G \rightarrow H$ is a group homomorphism, then f is onto if and only if $f(G) = H$. It is easily shown that if $f:G \rightarrow H$ is a homomorphism of groups and S is a subgroup of G, then f(S) is a subgroup of H. Hence images are important in group theory. If k1, k2, Z4, and j are defined as before, then the commands image(k1, Z4, j) and image (k4, Z4, j) return *[0]* and *[0, 3]* respectively. Note that both of these lists form subgroups of Z6. Also note that neither list is equal to Z6. Hence, k1 and k4 are not epimorphisms.

The Inverse Image of a Subset

Definition 2.9: Let $\phi:G \rightarrow H$ be a group homomorphism. Let S be a subset of H. The **inverse image** of S under ϕ is the set $\phi^{-1}(S) = \{ g \in G \mid \phi(g) \in S \}$.

The command invimage(S, f, G) finds the inverse image of a subset S of H where $f:G \rightarrow H$ is a group homomorphism. Note that the inverse image of $\{e_H\}$ is the kernel of f. If S is a subgroup of H, then it is easy to show that $f^{-1}(S)$ is a subgroup of G [Ga1 p. 195]. These are just a few reasons that inverse images are important. Suppose that k1, k4, and Z4 have been defined as before. The commands invimage([0], k1, Z4) and invimage([0], k4, Z4) return the lists *[0, 1, 2, 3]* and *[0, 2]* respectively. This is expected since these lists are the kernels of k1 and k4.

Isomorphisms

The final command in FGv1.mpl for morphisms is isom. This procedure checks to see if a mapping is a homomorphism. If the map is not a homomorphism, then the procedure returns a message stating this fact. Otherwise the procedure checks to see if the mapping is a monomorphism by seeing if the kernel is trivial. Then it checks to see if the mapping is onto by examining the image of the group being mapped. If the map is not one to one or onto, then *homomorphism* is returned. If the map is one to one but not onto, then *monomorphism* is returned. If the map is onto but not one to one, then *epimorphism* is returned. If the map is one to one and onto then *isomorphism* is returned.

It has already been shown that k1 and k2 as defined throughout this section are the only homomorphisms from Z4 to Z6. The commands isom(k1, Z4, j, Z6, g) and isom(k4, Z4, j, Z6, g) each

return *homomorphism*. This is no surprise since it has been shown that neither kernel is trivial, k1(Z4) ≠ Z6, and k4(Z4) ≠ Z6.

Define Z2:= [0, 1], o2:=(a, b)→(a+b) mod 2, Z6:=[0, 1, 2, 3, 4, 5], o6:=(a, b) → (a, b) mod 6, and f:= a → (3*a) mod 6. Then isom(f, Z2, o2, Z6, o6) returns *monomorphism*. Checking image(f, Z2, o2) gives *[0, 3]*. Hence Z_2 is isomorphic to the subgroup [0, 3] of Z_6. Now define f2:= a → a mod 2. Then isom(f2, Z6, o6, Z2, o2) returns *epimorphism*. Note that f2 is not one to one since ker f2 = [0, 2, 4].

Theorem 2.5 (1st Isomorphism Theorem): Suppose that f:G → H is a group homomorphism. Then there exists an isomorphism ϕ from G/ker(f) to f(G) such that ϕ(gKer(f)) = f(g).

Proof: See Gallian [Ga1 p.199].

Using the procedures in this chapter, one can demonstrate an example of the first isomorphism theorem found in Gallian [Ga1 p.199]. If one labels the vertices of a square one to four, then D_4, the set of all symmetries of a square, can be defined by the command pgp(4, {[[1, 2, 3, 4]], [[2, 4]]}). The eight permutations in this group correspond to four rotations and four reflections. One can define a map Phi from the group Pgp to itself as follows:

```
Phi:=proc(x)
 if(x=[] or x=[[1, 3],[2, 4]]) then RETURN([]); fi;
 if(x=[[1, 2, 3, 4]] or x=[[1, 4, 3, 2]]) then RETURN([[1, 4], [2, 3]]); fi;
 if(x=[[1, 4], [2, 3]] or x=[[1, 2], [3, 4]]) then RETURN([[1, 3], [2, 4]]); fi;
 RETURN([[1, 2], [3, 4]]);
end:
```

Then Homom(Phi, Pgp, mulperms, Pgp, mulperms) returns *homomorphism*. The command K:= kernel(Phi, Pgp, mulperms, Pgp, mulperms) returns the list *K = [[], [[1, 3], [2 ,4]]]* which is obvious from the definition of Phi.

Now one can define the factor group of K in Pgp as GMODH using the command GmodH(K, Pgp, mulperms, g). Recall that the kernel of a homomorphism is always a normal subgroup. Hence Isgp(GMODH, g) returns *Group*. By the first isomorphism theorem f:GMODH → f(Pgp) = [[], [[1, 4], [2, 3]], [[1, 3], [2, 4]], [[1, 2], [3, 4]]] given by f(a) = Phi(a[1]) should be an isomorphism. Testing this using isom(f, GMODH, g, [[], [[1, 4], [2, 3]], [[1, 3], [2, 4]], [[1, 2], [3, 4]]], mulperms) returns *isomorphism* as expected.

CONCLUSION

The algorithms behind these procedures are very simple and often inefficient for large groups. However, they allow the user a great deal of flexibility in defining groups. Many of the small groups that a student studies can be stored and manipulated using familiar notation. Something as simple as listing all of the subgroups of T_{12} could take an extremely large amount of time for a student to do by hand, but the algorithm subgps does it in a relatively small amount of time. Factor groups are quite confusing to many beginning algebra students since the group elements are cosets. The procedure GmodH allows one to store the cosets of a group and set up the operation to work with G/H. Many properties can be examined on the computer and elements listed that would be rather difficult to do with pen and paper. These are just a few reasons that the procedures in this chapter are useful.

Besides allowing the study of small finite groups, the procedures in this chapter provide a bridge to more complicated algorithms for handling larger groups. These procedures can be used to familiarize one with Maple and the concepts of storing groups on a computer. The next two chapters will deal with more complicated algorithms for handling particular types of groups.

CHAPTER 3: PERMUTATION GROUPS IN MAPLE

INTRODUCTION

In the last chapter, the author discussed many simple algorithms that work well with small finite groups but are inefficient for large groups. Over the last few decades, mathematicians have developed several clever algorithms for working with large groups of specific types. Two of the most successful areas of computational group theory have been the study of permutation groups and the study of groups defined by a finite representation of generators and relations. Maple's group package has procedures that work with both presentations. In this chapter, the author will discuss all of the procedures that relate to permutation groups and how they are implemented in Maple. The groups presented in terms of generators and relations will be discussed in the next chapter. Many of the procedures consist of two parts, one for each of the types of presentations. The two parts will be considered separately.

Since the symmetric group of degree n has order n!, the size of permutation groups can be quite large. Therefore, it is not practical to store all of the elements in the groups that one is considering. Fortunately, many questions about a group can be answered without ever storing each element. In fact, many questions can be answered simply by examining a small fraction of a group's elements. Some questions such as commutativity can be answered using only the set of generators. Many other group properties and subgroups can be discovered by examining what are known as stabilizer chains. These things will all be discussed in detail as Maple's procedures are explored one by one.

Representing Permutations

The first step in working with permutation groups is to somehow represent a permutation on a computer. There are two widely used notations for a permutation. One is cyclic notation, where the permutations are listed as a product of disjoint cycles. For example, (1, 2)(3, 4) is the permutation that swaps 1 and 2 and swaps 3 and 4. Many things, such as which elements are fixed by a permutation, can easily be determined by looking at a permutation in cyclic notation. However, cyclic notation does not easily transfer into computer language. It is easiest to store a permutation on {1, 2, …, n} by an n-tuple where the permutation sends i∈ {1, 2, …, n} to the i^{th} entry of the n-tuple. For instance, (1, 2)(3, 4) could be represented as [2, 1, 4, 3]. If a permutation in a group with large degree fixes several points, then cyclic notation is much shorter than the n-tuples, but basic calculations are much easier with the n-tuples. If a and b are two n-tuples, then the composition ba can be defined as a[b[i]] for i in {1, 2, …, n}. The inverse of an n-tuple a can be defined as the n-tuple c such that c[a[i]] = i for i from 1 to n. Both of these things can be done easily in most programming languages, making n-tuples superior to cycles in programming.

Maple uses a combination of n-tuples and products of disjoint cycles. There are procedures that convert between the two permutation types. For example, one can type convert([2, 1, 4, 3], 'disjcyc') at the Maple prompt to convert [2, 1, 4, 3] to [[1, 2], [3, 4]], which is Maple's notation for (1, 2)(3, 4). Conversely, one can type convert([[1, 2], [3, 4]], plist, 4) to convert [[1, 2], [3, 4]] to [2, 1, 4, 3]. The last

argument required to convert a product of disjoint cycles into an n-tuple is the degree of the group in which the permutation lies. For example, convert([[1, 2], [3, 4]], plist, 5) would return [2, 1, 4, 3, 5] which is in S_5 instead of [2, 1, 4, 3] in S_4.

As mentioned, cyclic notation has many advantages over n-tuples. For this reason, the procedures in the Maple group package use permutations written in cyclic notation for input and output, but Maple uses n-tuples inside the procedures to simplify computations. All of the procedures that contain a permutation in the input require that it be expressed as a product of disjoint cycles. Then the procedure converts the permutation into an n-tuple to be used by the algorithm. Any permutations that are to be returned as output are first converted back to cyclic form. Hence, the user may never know that n-tuples are being used.

Basic Computations

The procedures mulperms and invperm find the composition of two permutations and the inverse of a permutation respectively. If a and b are two permutations expressed as products of disjoint cycles, then the command mulperms(a, b) will return the composition of a and b as a product of disjoint cycles. The procedure converts a and b to n-tuples and then uses an internal routine `group/mp` to form the n-tuple p such that p[i] = b[a[i]] for i from 1 to n. Then p is converted to a product of disjoint cycles and returned to the user. The command invperm(a) converts a to an n-tuple and calls the internal routine `group/ip.` This routine finds the n-tuple c such that c[a[i]] = i for i from 1 to n. Then c is converted to a product of disjoint cycles for output. For example, invperm([[1, 2, 3]]) returns *[[1 ,3 ,2]]*, and mulperms([[1, 2, 3]], [[1, 3, 2]]) returns *[],* the identity cycle.

Defining a Permutation Group in Maple

Often one is more interested in working with an entire permutation group than just a few permutations. In Maple, this requires the use of the command permgroup. The user enters in the degree of the permutation group and a set of generators. Maple then performs some checks on the degree and generators to ensure that they define a group. For instance, the highest number appearing in any of the generators must be smaller than the degree. If there is a problem, an error message is returned. Otherwise, Maple sets up a function that defines the group with its degree and generators to be used by other commands. For example if one wants to work with S_4, then one choice of generators is (1, 2, 3, 4) and (1, 2). Thus one can type S4:= permgroup(4, {[[1, 2, 3, 4]], [[1, 2]]}) at the Maple prompt, and Maple will display *S4:= permgroup(4, {[[1, 2, 3, 4]], [[1, 2]]})*. On its own, the command is useless, but it is necessary to set up the groups that one wishes to use with other commands.

MAPLE'S PROCEDURES FOR PERMUTATION GROUPS

Maple has 22 commands in the group package for working with a permutation group, once it is defined with permgroup. They are invperm, mulperms, isabelian, orbit, grouporder, groupmember, RandElement, issubgroup, isnormal, cosets, cosrep, centralizer, normalizer, inter, center, areconjugate, NormalClosure, core, derived, DerivedS, LCS, and Sylow. The two commands mulperms and invperm have already been discussed. They work with individual permutations. The others work with permutation groups. As previously stated, it is impractical to work with every single element in a large permutation group. Hence, these commands make use of the generators and stabilizer chains to reduce computations.

Testing Commutativity

One of the simplest commands in the group package is isabelian. As stated in the last chapter, two elements a and b in a group are said to commute if ab = ba, and a group in which all of the elements commute with one another is called Abelian. Many permutation groups fail to have this property, including S_n for n >2. The procedure isabelian uses the following proposition to test a group for this property without calculating ab and ba for all a and b in the given group.

Proposition 3.1: Let G be a group with a set of generators gens(G). Then G is commutative if and only if xy = yx for all $x,y \in$ gens(G).

Proof: ($\Rightarrow$) Obvious. ($\Leftarrow$) Let $a,b \in G$. Then $a = x_1x_2...x_n$ and $b = y_1y_2...y_m$ where each $x_i,y_j \in$ gens(G) for $1 \le i \le n$ and $1 \le j \le m$. Then $ab = (x_1x_2...x_n)(y_1y_2...y_m) = (y_1y_2...y_m)(x_1x_2...x_n) = ba$ since xy = yx for all $x,y \in$ gens(G). ■

Hence, if a set of generators is known, one merely has to check that set for commutativity instead of the entire group. That is what the command isabelian(G) does. It tests all of the generators of G for commutativity. If one pair is found that does not commute then *false* is returned. Otherwise, *true* is returned, and the group is Abelian. For instance, if S4 has been defined as above, then isabelian(S4) will return *false* since S4 is not an Abelian group.

The Orbit of a Point

For an element i in the set I = {1, 2, ..., n}, the orbit of i under a group G is the set of all the points x in I such that p[i] = x for some permutation in G. Since all of the permutations in G are products of generators, the orbit of i can be found by using just the generators. To efficiently find the orbit of an integer i, Maple uses a queue and an orbit table. First, Maple stores a 1 in the i^{th} entry of the orbit table. Then Maple finds p[i] for all p in a set of generators of G. Any new integers are stored in a queue and a 1 is

stored in the slots corresponding to these integers in the orbit table. Then the procedure is repeated for each point in the queue each time storing any new points and discarding the checked element from the queue. Every time a new integer arises Maple stores a 1 in that entry of the orbit table. When the queue becomes empty, all the elements in the orbit have been tested. The orbit is the entries in the table that have a one. These are returned as a set of integers. For example, if PG := permgroup(48,{[[1, 3, 8, 6], [2, 5, 7, 4], [9, 33, 25, 17], [10, 34, 26, 18], [11, 35, 27, 19]], [[9, 11, 16, 14], [10, 13, 15, 12], [1, 17, 41, 40], [4, 20, 44, 37], [6, 22, 46, 35]], [[17, 19, 24, 22], [18, 21, 23, 20], [6, 25, 43, 16], [7, 28, 42, 13], [8, 30, 41, 11]], [[25, 27, 32, 30], [26, 29, 31, 28], [3, 38, 43, 19], [5, 36, 45, 21], [8, 33, 48, 24]], [[33, 35, 40, 38], [34, 37, 39, 36], [3, 9, 46, 32], [2, 12, 47, 29], [1, 14, 48, 27]], [[41, 43, 48, 46], [42, 45, 47, 44], [14, 22, 30, 38], [15, 23, 31, 39], [16, 24, 32, 40]]}), then orbit(PG, 1) returns the set of integers *{1, 3, 6, 8, 9, 11, 14, 16, 17, 19, 22, 24, 25, 27, 30, 32, 33, 35, 38, 40, 41, 43, 46, 48}*.

Stabilizer Chains

Proposition 3.2: Let G be a permutation group. The set $G_x = \{ g \in G \mid g[x] = x\}$ is a subgroup of G for each x in G.

Proof: Let x be an element of a permutation group G. Suppose that g and h are permutations in G_x. Then g[x] = x and h[x] = x. Since g[x] = x, $g^{-1}[x] = x$. Thus g^{-1} is in G_x for each g in G_x. Note that h[g[x]] = h[x] = x. Hence, $gh \in G_x$. Therefore, G_x is a subgroup of G by the two-step subgroup test [Ga1 p. 59]. ■

All of the commands in the group package for permutation groups, except for isabelian and orbit, require the use of a stabilizer chain. Let G be a permutation group acting on I = {1, 2, …, n}. Then for each $x \in I$, the stabilizer of x in G is the subgroup G_x defined in proposition 3.2. Note that G_x is the set of all permutations in G that fix x. One can select $y \in I \setminus \{x\}$ and find the stabilizer of y in G_x. This set $(G_x)_y = G_{x,y}$ is the set of all permutations in G that fix x and y. Since I is finite, this can be repeated until the stabilizer obtained fixes every element in I and hence is the identity. If one starts at G and repeatedly takes stabilizers of x_i and denotes $(G_{x_1 x_2 \ldots x_{i-1}})_{x_i} = G^{(i)}$ with $G = G^{(0)}$, then one will obtain the chain of stabilizers $G^{(0)} > G^{(1)} > \ldots > G^{(k)} = \{[\,]\}$. The sequence of integers $[x_1, x_2, \ldots, x_k]$ is known as the base for this stabilizer chain.

Stabilizer chains have been shown to have many nice properties. The most useful one in computational group theory is that a stabilizer of an element x in a group G has a nice set of coset representatives. Define trace(a,x) to be a permutation that maps x to a. Such elements exist for each a in the orbit of x. It is easy to see that the set $\{\text{trace}(a,x) \mid a \in \text{orb}(x)\}$ forms a complete set of coset representatives for G_x in G since each element of G_x fixes x. This gives a method for calculating a complete set of coset representatives for each $G^{(i)}$ in $G^{(i-1)}$ in a stabilizer chain. Many of Maple's procedures use such a set of coset representatives.

The most important internal routine used by Maple is `group/addperm.` This is the routine that sets up the stabilizer chains. From this point on, the term stabilizer chain will refer to the list output by `group/addperm` as opposed to simply referring to a chain of stabilizer subgroups. The input for `group/addperm` is a permutation and a stabilizer chain. The output is a new stabilizer chain for the group formed by adding the permutation to the generators of the old group, hence the name addperm. The first time that `group/addperm` is called for a group, the stabilizer chain input is [ ]. The routine uses this to form the stabilizer chain for the cyclic group generated by the permutation entered. The routine must be called recursively for each generator. The stabilizer chain formed consists of four parts. The first entry in the chain is a collection of generators of the original group $G = G^{(0)}$ and their inverses. The second entry is an element b_1 not fixed by the first generator entered into the stabilizer chain for G. This will be the first base element that will be fixed by $G^{(1)}$. The third entry is a complete set of coset representatives for $G^{(1)}$ in $G^{(0)}$ with their inverses. To be useful, these coset representatives are stored in a very precise manner. Recall that a complete set of coset representatives for G_x in G is $\{\text{trace}(a, x) \mid a \in \text{orb}(x)\}$. Maple allocates the maximum number of slots for coset representatives, which is the degree of the permutation group. Then [trace(a, b_1), `group/ip`(trace(a, b_1)] is stored in the a^{th} slot in the coset representatives. Any slots corresponding to integers not in the orbit of b_1 receive blank entries, [ ]. This provides Maple with a systematic way to search through the cosets. The final entry in the stabilizer chain is the stabilizer chain for $G^{(1)}$. Hence Maple stores a nested set of stabilizer chains which contains generators with inverses for each stabilizer, a base, and coset representatives with inverses for each $G^{(i)}$ in $G^{(i-1)}$. These three things together provide a powerful tool for studying a permutation group.

Each time that Maple calls `group/addperm,` it must perform several steps to construct the chain of stabilizers. First of all, the new permutation g and its inverse are added to the generators of $G^{(0)}$ as the pair $[g, g^{-1}]$. Then the coset representatives of $G^{(1)}$ in $G^{(0)}$ must be formed. To accomplish this, Maple uses a routine called `group/findorb.` Maple systematically takes products of previously calculated coset representatives, one of which is the identity, and the generators to find trace(a, b_1) for each a in the orbit of b_1 and stores them in the appropriate order. Then Maple must form the next link in the stabilizer chain. For this step, Schreier's lemma is used to find the generators for $G^{(1)}$. The lemma is as follows:

> Let $G = \langle g_1, g_2, \ldots, g_r \rangle$. Let H be a subgroup of index n in G, with coset representatives $x_1, x_2, \ldots, x_n$, where $x_1 = 1$ is the representative of H. Let $\overline{g}$ be the representative of the coset Hg. Then $H = \langle x_i g_j (\overline{x_i g_j})^{-1} \mid 1 \le i \le n, 1 \le j \le r \rangle$. [Ca p. 17]

Since Maple has coset representatives with inverses for $G^{(1)}$ in $G^{(0)}$, the lemma can be applied to find the generators of $G^{(1)}$. Maple does this calling `group/addperm` recursively with the appropriate generators until the chain is complete.

Maple actually uses a routine `group/stabchain` to form the stabilizer chain for a group. The routine calls `group/addperm` for each of the generators of the group. This sets up the complete stabilizer

chain for the group. All of the procedures in the group package that have yet to be discussed make some use of the stabilizer chain.

Finding the Size of a Group

One of the fundamental questions about a group is the number of elements that it contains. To answer this, Maple uses the procedure grouporder. Clearly, the order of $G^{(0)}$ is the product of the indices of each stabilizers $G^{(i)}$ in $G^{(i-1)}$ by LaGrange's Theorem. The index $|G^{(i-1)}:G^{(i)}|$ is the number of cosets of $G^{(i)}$ in $G^{(i-1)}$. Therefore, grouporder can calculate |G| simply by multiplying together the numbers of coset representatives for each stabilizer in the chain. As with most Maple procedures the work is actually done with an internal routine. The routine `group/chainsize` determines the size of a group defined by a particular stabilizer chain. Obviously, this is much more efficient than counting each element. One does not even have to know all of the group elements to find |G| using this method. For example if PG := permgroup(12,{[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]], [[1, 2]]}), then grouporder(PG) returns *479001600*.

Testing Membership

Another fundamental question about a group is whether or not it contains a particular permutation. To test if a permutation is in a group, Maple uses a procedure called groupmember. This procedure sets up the stabilizer chain for the group and then uses a routine called `group/inchain`. If G is a permutation group, then $g \in G$ implies that $g^{-1} \in G$. Thus, there exists a coset representative s_1 of $G^{(1)}$ in G such that $g^{-1}[b_1] = s_1[b_1]$ since the complete set of coset representatives is $\{\text{trace}(a, b_1) \mid a \in \text{orb}_G(b_1)\}$. Therefore, $g[s_1[b_1]] = b_1$, i.e., $s_1 g \in G^{(1)}$. This can then be repeated recursively until coset representatives s_i are found for i from 1 to k such that $s_k s_{k-1} \ldots s_2 s_1 g \in G^{(k)} = 1$. Hence $g = s_1^{-1} s_2^{-1} \ldots s_k^{-1} \in G$. That is how `group/inchain` tests membership. At each step in the chain of stabilizers, it looks for s_i such that $(s_{i-1} \ldots s_2 s_1 g)^{-1}[b_i] = s_i[b_i]$. If it fails at any step, then g is not in G, and *false* is returned. If it makes it all the way through, forming the identity, then $g \in G$, and *true* is returned. The power of this algorithm is that it only requires one to examine the coset representatives in the stabilizer chain instead of all of the elements in the group, making it quite efficient.

Finding Random Elements

The way that Maple performs grouporder and groupmember implies a way to generate a random element from a permutation group. A similar argument as used above in the membership test, replacing g^{-1} with g, shows that for any $g \in G$, there exist coset representatives such that $g = s_k s_{k-1} \ldots s_2 s_1$. Note that the order of G is the product of the numbers of coset representatives of $G^{(i)}$ in $G^{(i-1)}$ which is exactly the possible number of combinations of coset representatives $s_k s_{k-1} \ldots s_2 s_1$. Hence, there is a unique element of G for each of the possible combinations of coset representatives, and these are all of the elements of G. The procedure RandElement uses this fact to generate a random element. It selects a random coset

representative for each link in the stabilizer chain, by using Maple's random integer generator, and then it multiplies these together forming a unique element of G.

Subgroups

Given two permutation groups of the same degree, one might desire to know if one of the two is a subgroup of the other. Since both groups are well defined with the same operation, one merely needs to test if all of the elements of one group lie in the other. Furthermore, since every element is a product of generators, it is only necessary to see if all of the generators of one group lie in the other. That is all that the routine issubgroup does. It tests to see if the generators of a permutation group H lie in a permutation group G by forming the stabilizer chain for G and using `group/inchain.` Note that as previously stated, Maple's procedures use permutation lists instead of products of disjoint cycles for computations. Hence, issubgroup uses `group/inchain` to test membership instead of groupmember, which takes products of disjoint cycles, converts them to permutation lists, and calls `group/inchain.`

Testing Normality

Once a subgroup of a permutation group is known, one might want to know if it is a normal subgroup. A subgroup H is normal in a permutation group G if $xsx^{-1} \in H$ for all s in H and all x in G. To verify this statement would require working with all of the elements in both of the groups, which is very inefficient. A better method is to use proposition 3.3.

Proposition 3.3: Let H be a subgroup of a group G, and let gens(H) and gens(G) be the generators of H and G respectively. Then H is normal in G if and only if $ghg^{-1} \in H$ for all $g \in$ gens(G) and all $h \in$ gens(H) .

Claim 3.1: Let H be a subgroup of a group G. Let t be a positive integer. Suppose that $\{h_1, h_2, \ldots, h_t\}$ is a subset of H, $x \in G$, and $xh_ix^{-1} \in H$ for $1 \le i \le t$. Then $x(h_1h_2 \ldots h_t)x^{-1} \in H$.

Proof: Obviously, the claim is true for t = 1. Suppose that the claim is true for some t > 1. Note that $x(h_1h_2 \ldots h_th_{t+1})x^{-1} = x(h_1h_2 \ldots h_tx^{-1}xh_{t+1})x^{-1} = (x(h_1h_2 \ldots h_t)x^{-1})(xh_{t+1}x^{-1})$. By the induction hypothesis, $(x(h_1h_2 \ldots h_t)x^{-1}) \in H$. By assumption, $(xh_{t+1}x^{-1}) \in H$. Hence, $x(h_1h_2 \ldots h_th_{t+1})x^{-1} \in H$ by closure. Therefore, the claim holds for any positive integer t by the principle of mathematical induction [Ga1 p. 13]. ■

Claim 3.2: Let H be a subgroup of G. Let m be a positive integer. Suppose that $\{x_1, x_2, \ldots, x_m\}$ is a subset of G and $x_ihx_i^{-1} \in H$ for $1 \le i \le m$ for each $h \in H$. Then $(x_1x_2 \ldots x_m)h(x_1x_2 \ldots x_m)^{-1} \in H$ for each $h \in H$.

Proof: Obviously, the claim is true for m = 1. Suppose that the claim is true for some m > 1. Note that $(x_1x_2 \ldots x_mx_{m+1})h(x_1x_2 \ldots x_mx_{m+1})^{-1} = (x_1x_2 \ldots x_mx_{m+1})h(x_{m+1}^{-1}x_m^{-1} \ldots x_2^{-1}x_1^{-1}) =$

$(x_1x_2 \ldots x_m)(x_{m+1}hx_{m+1}^{-1})(x_m^{-1} \ldots x_2^{-1}x_1^{-1})$. By assumption, $x_{m+1}hx_{m+1}^{-1} = h_2$ for some h_2 in H. So $(x_1x_2 \ldots x_mx_{m+1})h(x_1x_2 \ldots x_mx_{m+1})^{-1} = (x_1x_2 \ldots x_m)h_2(x_m^{-1} \ldots x_2^{-1}x_1^{-1}) = (x_1x_2 \ldots x_m)h_2(x_1x_2 \ldots x_m)^{-1} \in H$ by induction hypothesis. Therefore, the claim is true for all positive integers m by the principle of mathematical induction [Ga1 p. 13]. ■

Proof of Proposition 3.3: ($\Rightarrow$) If H is normal in G, then $ghg^{-1} \in H$ for all $g \in G$ and all $h \in H$. ($\Leftarrow$) Suppose that $aba^{-1} \in H$ for all $a \in$ gens(G) and all $b \in$ gens(H). Let $h \in H$ and $x \in G$. Then $h = h_1h_2 \ldots h_t$ for some combination of elements in gens(H). By claim 3.1, $xhx^{-1} = xh_1h_2 \ldots h_tx^{-1} \in H$ provided that $xh_ix^{-1} \in H$ for $1 \le i \le t$. Note that $x = x_1x_2 \ldots x_m$ for some combination of elements in gens(G). By claim 3.2, $xh_ix^{-1} = (x_1x_2 \ldots x_m)h_i(x_1x_2 \ldots x_m)^{-1} \in H$ for $1 \le i \le t$ provided that $x_jh_ix_j^{-1} \in H$ for $1 \le j \le m$ and $1 \le i \le t$. Hence, $xh_ix^{-1} \in H$ for $1 \le i \le t$ since $x_jh_ix_j^{-1} \in H$ for $1 \le j \le m$ and $1 \le i \le t$ by hypothesis. Therefore, $xhx^{-1} \in H$. Hence, H is a normal subgroup of G. ■

The procedure isnormal tests the property of proposition 3.3 using `group/inchain` on each aba^{-1} where $a \in$ Gens(G) and $b \in$ Gens(H). If one aba^{-1} is found that is not in H, then *false* is returned. Otherwise, *true* is returned.

Cosets

The use of the stabilizer chain already shows some of the significant uses of cosets. Maple has a routine cosets to find a set of coset representatives for a subgroup of a permutation group. Let G be a permutation group acting on the set I = {1, 2, ..., n}. For any element p in G and any s in a subgroup H, ps is a coset representative for p with respect to H. To be useful, one needs a way to always select the same element in a coset as the representative. For this, one needs some way to order permutations. If the list $[b_1, b_2, \ldots, b_k]$ is a base for G, then the base image of an element g in G is the list of integers $[g[b_1], g[b_2], \ldots, g[b_k]]$. Note that $g[b_i] = h[b_i]$ for $1 \le i \le k$ implies that g and h lie in the same coset of $G^{(k)}$ in $G^{(k-1)}$. Thus $g[b_i] = h[b_i]$ for $1 \le i \le k$ implies that g = h since $G^{(k)} = \{e\}$. Therefore, base images are unique for each $g \in G$. We will demonstrate that permutations may be ordered lexicographically by their base images. Before we can assign an ordering to the elements of G, we need to use the base to assign an ordering to the elements of I. Let x and y be elements of I. If x and y are in the base, then we will say $x < y$ with respect to the base if and only if $x = b_i$, $y = b_j$, and $i < j$. If x is in the base and y is not, then we will say that $x < y$ with respect to the base. Finally, if neither x nor y is in the base, then we will say that $x < y$ with respect to the base if and only if $x < y$ as real number. Using this method to order the elements of I, one can order the base images of the elements of G. We will say that $[g[b_1], g[b_2], \ldots, g[b_k]]$ is smaller than $[h[b_1], h[b_2], \ldots, h[b_k]]$ if $[g[b_1], g[b_2], \ldots, g[b_i]] = [h[b_1], h[b_2], \ldots, h[b_i]]$ and $g[b_{i+1}] < h[b_{i+1}]$ with respect to the base. For example, if a permutation group with a stabilizer chain has a base [2, 3, 1, 4, 5], then the base

image [2, 3, 5, 4, 1] < [2, 1, 3, 4, 5].

Let H be a subgroup of G, and let $g \in G$. Consider the right coset $Hg = \{hg \mid h \in H\}$. Note that one can select any of the elements of Hg as a coset representative for Hg in G. It would be nice to denote a particular element as the coset representative for a given coset. Since the base image of an element is unique, there exists a unique element in every coset with the smallest base image. Hence, one can define the representative of a coset to be this unique element. This ensures that the same element will always be selected as a coset representative.

Proposition 3.4: Let H be a subgroup of a group G. Let $g \in G$. Let q be the element of Hg with the smallest base image. If H has a stabilizer chain $H = H^{(0)} > H^{(1)} > H^{(2)} > ... > H^{(k)} = \{e\}$, then there exist coset representatives c_i for $H^{(i)}$ in $H^{(i-1)}$ such that $q = c_k c_{k-1} ... c_2 c_1 p$ for any p in Hg.

Proof: Since q and p are both elements of Hg, Hq = Hp. Hence, $qp^{-1} \in H$ [Ga1 p. 133]. As previously shown, $qp^{-1} \in H$ implies that there exist coset representatives c_i for $H^{(i)}$ in $H^{(i-1)}$ for $1 \le i \le k$ such that $qp^{-1} = c_k c_{k-1} ... c_2 c_1$. Hence, $q = c_k c_{k-1} ... c_2 c_1 p$. ■

Maple's internal routine `group/cosetrep` takes advantage of proposition 3.4 to find the unique element of Hp that has the smallest base image for any p in G. Note that this ensures that two different elements in the same coset will lead to the same representative. This is essential for algorithms that involve checking properties on entire cosets instead of elements.

By proposition 3.4, the element with the smallest base image in a coset can be found by choosing the appropriate coset representatives c_i for $H^{(i)}$ in $H^{(i-1)}$ and letting $q = c_k c_{k-1} ... c_2 c_1 p$. To accomplish this, Maple works its way through a chain of stabilizers. At the j^{th} step, Maple must select c_j such that the element $c_j c_{j-1} ... c_2 c_1 p$ has as small a base image as possible. Recall that the j^{th} link in a stabilizer chain for H formed by Maple contains a collection of generators for $H^{(j-1)}$, the j^{th} base element b_j, a complete set of coset representatives for $H^{(j)}$ in $H^{(j-1)}$, and a stabilizer chain for $H^{(j)}$. The set of coset representatives is stored as a list. Recall that a complete set of coset representatives for $H^{(j)}$ in $H^{(j-1)}$ can be found by forming trace(a, b_j) for each a in the orbit of b_j under $H^{(j-1)}$. Maple stores trace(a, b_j) in the a^{th} slot of the coset list in the chain. Any slots corresponding to integers not in the orbit of b_j are assigned [ ]. Maple must select c_j from the elements in the list that are not [ ]. Since each coset representative for $H^{(j)}$ in $H^{(j-1)}$ fixes the base elements $b_1, b_2, ..., b_{j-1}$, being in the stabilizer of those elements, $(xc_{j-1} ... c_2 c_1 p)[b_i] = (c_{j-1} ... c_2 c_1 p)[b_i]$ for $1 \le i < j$ for each x in this list of coset representatives. Hence, to find the appropriate c_j, one need only ensure that $(c_j c_{j-1} ... c_2 c_1 p)[b_j]$ is as small as possible. To find this c_j, Maple uses a loop to calculate the integer m such that $(c_{j-1} c_{j-2} ... c_2 c_1 p)[m]$ is the smallest element of I with respect to the base and m is in the orbit of b_j under $H^{(j-1)}$. Then c_j is selected to be the m^{th} entry in the list of coset representatives for $H^{(j)}$ in $H^{(j-1)}$. Hence, $(c_j c_{j-1} ... c_2 c_1 p)[b_j] = (c_{j-1} ... c_2 c_1 p)[c_j[b_j]] = (c_{j-1} c_{j-2} ... c_2 c_1 p)[m]$. By selecting c_j in this manner for $1 \le j \le k$,

one ensures that $c_k c_{k-1} \ldots c_2 c_1 p$ is the unique element of Hp with the smallest base image.

The procedure cosets is used to find the complete set of coset representatives for a subgroup H of a group G. Most of the work is done by the internal routine `group/cosets1` which uses `group/cosetrep` to construct a complete set of coset representatives for H in G. The procedure `group/cosets1` is outlined as follows:

routine: `group/cosets1`
input: an element x of G; a set of generators gens(G); a set S of coset representatives for H in G; H
output: a possibly modified set of coset representatives S
1. Let r:=`group/cosetrep`(x, H), the coset representative for Hx with the smallest base image.
2. If the r is all ready in S, then return S.
3. Let S:= S union {r}.
4. For each generator g in gens(G) do
   S:= `group/cosets1`(rg, gens(G), S)
  enddo
5. Return S.

The procedure cosets converts the generators of G to n-tuples and forms the stabilizer chain for H. Then it calls `group/cosets1` with x equal to the identity, e = [1, 2, ..., n]. Proposition 3.5 shows that the procedure works recursively through products of generators and known coset representatives to form a complete set of coset representatives for H in G.

Proposition 3.5: If G is a finite group, then the procedure `group/cosets1`(e, gens(G), {}, H) will terminate and return a complete set of coset representatives for H in G.

Proof: Obviously, the routine is recursive. First we will show that recursion cannot occur indefinitely, and the procedure must terminate. Note that each time `group/cosets1`(x, gens(G), S, H) is executed, there are three possibilities:

1. The coset representative r of Hx is in S, which causes `group/cosets1` to return S.
2. The coset representative r of Hx is not in S. The element r is added to S, and rg is in the new S for each g in gens(G). This completes the routine for that particular x, and S is returned.
3. The coset representative r of Hx is not in S. The element r is added to S, and rg is not in the new S for some g in gens(G). This leads to another level of recursion.

Note that deeper recursion only occurs with case 3, and every time that this occurs, a new element is added to S. Since G is finite, there are finitely many coset representatives for H in G. Hence, there are only a finite number of elements that can be added to S. So case 3 cannot occur indefinitely. Hence, each recursive level

must terminate. Thus, the overall routine must terminate. It remains to show that the procedure will find a complete set of coset representatives for H in G. Note that $a \in G$ implies that $a = a_1a_2 \ldots a_t$ for some combination of not necessarily distinct a_i's in gens(G). Hence, a right coset of H in G is of the form $Ha_1a_2 \ldots a_t$ where each a_i is in gens(G). We will use induction on t to prove that for each element in G of the form $a_1a_2 \ldots a_t$, the command `group/cosets1`(e, gens(G), {}, H) calls `group/cosets1`(x, gens(G), S, H) for some x in $Ha_1a_2 \ldots a_t$. Thus, we will show that an element is checked from each coset, and a complete set of coset representatives will be found. Let t = 1. When `group/cosets1`(e, gens(G), {}, H) is executed, case 1 occurs since e is the coset representative of He ,and e is not in {}. Hence, the routine must call `group/cosets1`(eg = g, gens(G), S, H) for each g in gens(G). This checks at least one element from each coset of the form Ha_1. Now suppose that for some $t > 1$, `group/cosets1`(x, gens(G), S, H) is called for some x in $Ha_1a_2 \ldots a_t$ for each $a_1a_2 \ldots a_t$ in G. Observe that a coset of the form $Ha_1a_2 \ldots a_ta_{t+1} = (Ha_1a_2 \ldots a_t)a_{t+1}$. By the induction hypothesis, the procedure calls `group/cosets1`(x, gens(G), S) for an element in $Ha_1a_2 \ldots a_t$. The first time that this occurs, the coset representative r of $Ha_1a_2 \ldots a_t$ is added to S, and the routine must call `group/cosets1`(rg, gens(G), S, H) for each g in gens(G). Since $Hr = Ha_1a_2 \ldots a_t$, $Hrg = Ha_1a_2 \ldots a_tg$ for each g in gens(G). Therefore, some element in each coset of the form $Ha_1a_2 \ldots a_ta_{t+1}$ must be checked. So by the principle of mathematical induction [Ga1 p. 13], every coset is checked. Hence the algorithm will return a complete set of coset representatives for H in G. ■

The Procedure cosrep

Since a group is the disjoint union of the cosets of any subgroup {Hu p. 38], for each $g \in G$, there exists a coset representative and an element in the subgroup that multiply to give g. Maple finds these with the command cosrep. The procedure uses `group/cosetrep` to get the coset representative c for a permutation p. The subgroup element is then defined as p times the inverse of its coset representative. Cleary $(pc^{-1})c = p$, and the cancellation law ensures that pc^{-1} is the unique subgroup element. Hence pc^{-1} and c are the desired elements.

Finding a Centralizer, Normalizer, or Intersection

Suppose that H is a subgroup of a group G defined by a property such as the centralizer of a set, normalizer of a subgroup, or the intersection of two subgroups. Suppose that K is a known subgroup of H. If $a \notin H$ for some $a \in G$, then $ak \notin H$ for all $k \in K$. Otherwise, $k \in K \subset H$ implies that $k^{-1} \in H$ which implies that $(ak)k^{-1} = a \in H$. This fact can be used to find H without examining each element in G for the given property. To find such subgroups, Maple uses a routine `group/findprop.` This routine uses `group/findprop1` and `group/addperm` to form the subgroup that satisfies the desired property. Given a stabilizer chain for a subgroup K of the desired subgroup, `group/findprop1` finds another element a in the desired subgroup by searching coset representatives similarly to the method used by `group/cosets1.` The routine `group/addperm` then creates the stabilizer chain for $\langle K, a \rangle$. The routine `group/findprop` starts by calling

findprop1 with K:={[]}.Then it repeatedly uses `group/findprop1` and `group/addperm` until all of the elements in the desired subgroup have been found.

To find all the elements with a given property, one could check every single element in the group. However, this is not practical for large groups. The power of constructing the subgroups one generator at a time, as `group/findprop` does, is that one can add many elements at once and throw out entire cosets of the known subgroup K. Some elements actually get checked more than once when `group/findprop1` is called repeatedly, but as the cosets grow larger and larger, the number of coset representatives shrinks dramatically. Thus for large groups, the desired subgroup may be found by checking a small fraction of the elements.

The procedures centralizer, normalizer, and inter all define a property and call `group/findprop` to form the subgroup of all elements satisfying that property. Let G be a group and $x \in G$. If C is a subset of G, then the property defined by centralizer is $xc = cx$ for all $c \in C$. If H is a subgroup of G, then the property defined by normalizer is $xhx^{-1} \in H$ for all $h \in H$. If H and K are two subgroups of G, then the property defined by inter is $x \in H \cap K$. It was pointed out in chapter two that centralizers and normalizers are subgroups. An easy application of the one-step subgroup test [Ga1 p. 58] shows that the intersection of two subgroups is a subgroup since a, $b \in H$ and a, $b \in K$ imply that $ab^{-1} \in H$ and $ab^{-1} \in K$.

Testing for Conjugates

The procedure areconjugate uses `group/findprop` to determine whether or not two permutations g_1 and g_2 are conjugates of one another. The routine `group/findprop` calls `group/findprop1` with a special argument Not_a_sbgr. Then `group/findprop1` searches through elements of the group looking for a p such that (g1)(p)=(p)(g2). If one is found, *p,true* is returned to `group/findprop.` The *true* tells `group/findprop` not to look for any other elements, and it returns *p,true* to areconjugate. If only three arguments are used, areconjugate then returns *true*. For example, if one has defined G:= permgroup(4, {[[1, 2, 3, 4]], [[1, 2]]}), then areconjugate(G, [[2, 3]], [[1, 4]]) returns *true.* If Not_a_sbgr is entered as a forth argument, Maple returns *true* and the element p. For instance, areconjugate(G, [[2, 3]], [[1, 4]], Not_a_sbgr) returns *true, [[1, 3], [2, 4]]* since ([[1, 3], [2, 4]])([[2, 3]])([[1, 3],[2, 4]])$^{-1}$ = [[1, 4]]. If `group/findprop1` finds no element p, then *[ ]* is returned, and areconjugate returns *false*.

The Center of a Group

The center of a group is the set of all elements that commute with every element in the group. Obviously, this is equivalent to the set of all elements that commute with each of the generators of the group. Hence, the center of a group is just the centralizer of the generators. Thus the procedure center uses the procedure centralizer on the generators. Quite often, the center of a large non-Abelian group is trivial, and the centralizer of a small number of its generators is trivial. For this reason, Maple uses a recursive or algorithm that finds the centralizer of a subset of the generators, adding one at a time until they are all used

the trivial group is obtained.

Normal Closures

The normal closure of a subgroup H of a group G is the smallest normal subgroup of G that contains H. By proposition 3.3, a subgroup H of a group G is normal if $ghg^{-1} \in H$ for all g in the generators of G and all h in the generators of H. If N is the normal closure of H in G, then N is normal in G. Hence $ghg^{-1} \in N$ for all $g \in$ gens(G) and all $h \in$ gens(H). Maple uses this fact to construct the normal closure N of a subgroup H of a permutation group G. The procedure NormalClosure sets lg equal to H. Then the algorithm checks to see if $aba^{-1} \in$ lg for each $a \in$ gens(G) and each $b \in$ gens(lg). If all such products are in lg, then lg is the normal closure of H. Otherwise, any aba^{-1} not in lg is added to the end of the list of generators of lg, and the procedure is iterated until $aba^{-1} \in$ lg for all $a \in$ gens(G) and all $b \in$ gens(lg). Note that at each iteration there are two possibilities. First of all, the algorithm could find that $aba^{-1} \in$ lg for all $a \in$ gens(G) and all $b \in$ gens(lg). If this occurs then lg is a normal subgroup of G. Since lg is formed by adding only the elements necessary to create a normal subgroup, lg must be the normal closure of H in G. The second possibility is that the algorithm finds an element $aba^{-1} \notin$ lg which must be added. Obviously, this cannot occur indefinitely since G is finite. Hence, the procedure must terminate finding the normal closure of H.

The Core of a Subgroup

The core of a subgroup H in a permutation group G is the largest normal subgroup of G contained in H. Maple finds the core of H in G with the command core(H, G). This procedure uses some of the ideas similar to those used to construct the normal closure. First, Maple sets csg = H. Then one at a time Maple tests against the generators of G to see if $aba^{-1} \in$ csg for all the generators b of csg and the generator a of G. If one is found such that aba^{-1} is not in csg, then Maple sets up the generators of a subgroup M of G such that $M=\langle aba^{-1} | b \in \text{Gens(csg)} \rangle = a(\text{csg})a^{-1}$. Then csg is set to equal csg $\cap$ M, and the procedure is repeated using the new csg. Note that if N<H such that N is normal in G, then $gNg^{-1} = N$ for all $g \in$ gens(G). Thus, for each $n \in N$ and each $g \in$ gens(G), there exists an n' in N such that $gn'g^{-1} = n$. It follows that any normal subgroup of csg must be contained in csg $\cap[a(\text{csg})a^{-1}]$ for each generator of G. Therefore, the procedure used by Maple must reach the largest normal subgroup of G contained in H, at which point the routine terminates, and the core is returned.

The Derived Subgroup

There are many definitions of a solvable group. Each of these has its own significance in different areas of group theory. One definition as stated in Hungerford [Hu p. 102] is based on a chain of derived subgroups. If G is a group, then the derived subgroup (or commutator subgroup) of G is the group $G' = \langle \{aba^{-1}b^{-1} | a,b \in G\} \rangle$. Once G' is found, one can construct $(G')' = \langle \{aba^{-1}b^{-1} | a,b \in G'\} \rangle$. Denote G = G(0), G(1) =

G′, and G(i) = (G(i-1))′. Then there is a chain of derived subgroups G(0)≥G(1)≥G(2)≥ … called the derived series of G. Eventually, since G is finite, G(i) will equal G(j) for all j>i for some i, and the chain will terminate. A group is solvable if its derived series terminates with the trivial group {e}. Maple can construct the derived subgroup of a group. This, in turn, can be used to construct the derived series of a group, which tells the user whether or not a group is solvable.

Proposition 3.6: Let G be a group with a normal subgroup H. Define [H, G] to be the subgroup $\langle hgh^{-1}g^{-1} \mid h \in H, g \in G \rangle$. Then [H, G] is equal to the normal closure of the subgroup K = $\langle hgh^{-1}g^{-1} \mid h \in gens(H), g \in gens(G) \rangle$.

Proof: First, we will show that [H, G] is normal in G. Let $x \in G$ and $y \in [H, G]$. Then $xyx^{-1} = x(hgh^{-1}g^{-1})x^{-1}$ for some $h \in H$ and $g \in G$. Hence $xyx^{-1} = (xhx^{-1})(xgx^{-1})(xh^{-1}x^{-1})(xg^{-1}x^{-1})$. Since H is normal in G, $xhx^{-1} = h'$ for some $h' \in H$. Since $x,g \in G$, $xgx^{-1} = g'$ for some $g' \in G$. Since $(aba^{-1})^{-1} = (a^{-1})^{-1}b^{-1}a^{-1} = ab^{-1}a^{-1}$ for all a and b in G, $(h')^{-1} = xh^{-1}x^{-1}$ and $(g')^{-1} = xg^{-1}x^{-1}$. Hence, $xyx^{-1} = h'g'(h')^{-1}(g')^{-1} \in [H, G]$. Therefore, [H, G] is normal in G. It follows that the normal closure of K is contained in [H, G] since $K \subseteq [H, G]$ and the normal closure of K is the smallest normal subgroup of G containing K. It remains to show that [H, G] is contained in the normal closure of K. To simplify notation, let $hgh^{-1}g^{-1} = [h, g]$. Note that $a[b, c]a^{-1}[a, c] = a(bcb^{-1}c^{-1})a^{-1}(aca^{-1}c^{-1}) = abcb^{-1}a^{-1}c^{-1} = (ab)(c)(ab)^{-1}(c)^{-1} = [ab, c]$, and $[a, b]b[a, c]b^{-1} = (aba^{-1}b^{-1})b(aca^{-1}c^{-1})b^{-1} = abca^{-1}c^{-1}b^{-1} = (a)(bc)(a)^{-1}(bc)^{-1} = [a, bc]$ for all a,b, and c. An arbitrary element of [H, G] is of the form $[a_1a_2\ldots a_n, b_1b_2\ldots b_m]$ where $n,m \in \mathbf{N}$ and each $a_i \in gens(H)$ and each $b_i \in gens(G)$. The normal closure of K by definition contains all elements of the form $xaba^{-1}b^{-1}x^{-1}$ where $x \in G$, $a \in gens(H)$, and $b \in gens(H)$. Then $[a_1a_2, b_1] = a_1[a_2, b_1]a_1^{-1}[a_1, b_1]$ and $[a_3, b_2b_3] = [a_3, b_2]b_2[a_3, b_3]b_2^{-1}$ are in the normal closure of K for all a_1, a_2, and a_3 in gens(H) and all b_1, b_2, and b_3 in gens(G) . It then follows by induction that $[a_1a_2\ldots a_n, b_1b_2\ldots b_m]$ is in the normal closure of K for an arbitrary element in [H, G]. Hence, [H, G] is contained in the normal closure of K. So [H, G] is the normal closure of K.■

Maple uses the command derived(G) to construct the derived subgroup of a permutation group G. Note that the derived subgroup of G is $\langle \{aba^{-1}b^{-1} \mid a, b \in G\} \rangle = [G, G]$. Using the definition would require calculating the commutator, $aba^{-1}b^{-1}$, for all a and b in G. This would be very costly if the order of G is large. However, by proposition 3.6, one can calculate the commutators for the generators of G and take the normal closure. Hence, the procedure derived computes all of the distinct commutators of the generators of G and then finds the normal closure using NormalClosure.

A Derived Series

The Maple command DerivedS(G) recursively calls derived to form the derived series of G. When two successive derived subgroups have the same order, the process terminates. If the final derived subgroup in the series is permgroup(n, [ ]), the trivial group, then G is a solvable group.

Lower Central Series

A similar concept to a solvable group is a nilpotent group. Just like solvable groups, there are several ways to define a nilpotent group. One way uses a lower central series. In [Ro p.151] Rose defines a lower central series and an upper central series as follows:

> We define subgroups $\Gamma_n(G)$ and $Z_n(G)$ of G recursively as follows. Let $\Gamma_1(G) = G$ and $Z_0(G) = 1$. Then, for each integer $n>1$, $\Gamma_n(G) = [\Gamma_{n-1}(G), G]$, and for each integer $n>0$, $Z_n(G)/Z_{n-1}(G) = Z(G/Z_{n-1}(G))$. Then
>
> $$G = \Gamma_1(G) \geq \Gamma_2(G) \geq \Gamma_3(G) \geq \ldots \qquad \text{(a)}$$
>
> and $$1 = Z_0(G) \leq Z_1(G) \leq Z_2(G) \leq \ldots \qquad \text{(b)}$$
>
> The descending sequence (a) is called the lower central series of G and the ascending sequence (b) is called the upper central series of G.

Rose goes on to show that it is equivalent to define a group to be nilpotent if either the lower central series terminates at {e} or the upper central series terminates at G [Ro p.152-153].

Maple has a procedure LCS that computes the lower central series of a permutation group, which can be used to determine if a group is nilpotent. LCS recursively finds the subgroups Γ_n by forming the subgroup $H_n = \langle \{aba^{-1}b^{-1} \mid a \in \text{Gens}(\Gamma_{n-1}) \text{ and } b \in \text{Gens}(G)\} \rangle$ and then finding the normal closure of H_n. Proposition 3.6 and its proof assure that this is a valid computation of Γ_n. The procedure terminates when two consecutive subgroups have the same order. If the last entry in the series is {e}, then the group is nilpotent.

Sylow Subgroups

Let p be a prime. The first Sylow theorem asserts that a group of order $p^n m$ where gcd(p, m) = 1 has subgroups of order p^i for i from 1 to n and each subgroup of order p^i is normal in a subgroup of order p^{i+1} [Hu p. 94]. A Sylow p-subgroup is one of order p^n. Much research has been done for finding Sylow p-subgroups of large groups. Some of the most productive methods have been using a divide and conquer algorithm on homomorphic images described in [Bu]. Maple uses a widely known method of searching centralizers of elements of order p. Consider the following lemma from [BC]:

Lemma: Suppose H is a normal subgroup of K and $K = \langle H, x \rangle$. If t is the smallest positive integer such that $x^t \in H$, then $|K| = |H| \times t$.

Suppose that one has a subgroup P_i of order p^i and wants to find a subgroup P_{i+1} of order p^{i+1} such that P_i is normal in P_{i+1}. Then $P_{i+1} = \langle P_i, x \rangle$ for some x since otherwise there would be a subgroup H such that $|H| = n \neq p^i$, $n \neq p^{i+1}$, and $p^i | n | p^{i+1}$. Since P_i is normal in P_{i+1}, such an x must lie in the normalizer of P_i in G. It follows from the lemma that x is not an element of P_i, but $x^p \in P_i$. Therefore if one desires to construct P_{i+1} from P_i, it is only necessary to find such an x. This can be repeated until the Sylow p-subgroup is found.

It is much easier to find centralizers than normalizers. Hence an algorithm has been devised to reduce the size of the group to be normalized by searching centralizers. A detailed description of this method is contained in [BC]. Butler and Cannon note that if P is a p-subgroup of G such that $|P| = p^i$ and p^{i+1} divides |G|, then there exists an element z, of order p, in Z(P) such that p^{i+1} divides $|C_G(z)|$. Thus the centralizer contains a subgroup of order p^{i+1}, and the desired x lies in $C_G(z)$. Also since $|C_G(z_1)| = |C_G(z_2)|$ for conjugates z_1 and z_2, it is only necessary to search conjugacy class representatives for z. This is the algorithm that Maple uses in Sylow. It finds an element a with order p^ij for some $i,j \in \mathbf{N}$. Then it forms $P := \langle a^j \rangle$, a subgroup of order p^i. This gives a p-subgroup P to start with. Then Z(P) is searched for conjugacy class representatives of order p. When one is found, a subroutine `group/cyc_ext` is called. It forms the normalizer of P in $C_G(z)$. Then it searches for an element such that $x \notin P$ but $x^p \in P$. Then x is added to the generators of P and the process is repeated until $|P| = p^n$, and the desired subgroup is found.

CONCLUSION

In the previous chapter, the author discussed generalized algorithms for small finite groups. These algorithms are able to do many of the things mentioned in this chapter. However, being written for general groups, they often require checking properties for every single element in a group. This can take a very long time with large groups. Even the fast processors on the market today would require a long time to check a property on 40! elements. Therefore, the procedures in this section are far superior for permutation groups. Cayley's theorem [Ga1 p. 119] asserts that every group is isomorphic to a group of permutations. So in theory, one could study any finite group on a computer as a permutation group. Many real world problems involve the arrangement of objects and hence permutations. For these reasons, the study of permutation groups is an important area of group theory. The routines in Maple's group package are some of the most efficient algorithms currently known for studying these groups.

CHAPTER 4: FINITELY PRESENTED GROUPS

In 1936 Todd and Coxeter published a method for finding the index of a subgroup of a group that is represented by a finite presentation of generators and relations [TCx]. This method bearing their names has been shown to provide a great deal of information about a group. Besides finding the index of a subgroup, it can be used to do such things as find coset representatives, calculate group order, and form a representation of the group as a permutation group. The last is quite significant given the efficient methods known for working with permutation groups as described in the last chapter. This chapter will discuss the version of the Todd-Coxeter algorithm performed by Maple and the procedures in the group package that use it. Maple actually performs a modified version of the algorithm to obtain more information than just the coset numbers and representatives. This chapter will describe the basic algorithm and the modifications separately. Throughout this chapter the Todd-Coxeter algorithm will be referred to as the TC algorithm for convenience.

INTRODUCTION TO THE TC ALGORITHM

Let G be a group given by the finite presentation $G = \langle g_1, g_2, ..., g_n \mid R_1 = R_2 = ... = R_m = 1 \rangle$ where the g_i's are generators of G and the R_i's are words in the generators and their inverses. For the remainder of this chapter, a word in the generators of G and their inverses will be referred to simply as a word in the generators of G. Let $H = \langle h_1, ..., h_s \rangle$ where each h_i is a word in the generators of G. Suppose that [G:H] = k. Denote the cosets of H in G with the integers 1 to k, beginning with H = 1. The idea of the TC algorithm is to construct a coset table that defines multiplication between each coset and each generator and between each coset and each inverse of a generator of G. Hence if G has n generators and k cosets, then a coset table for G is a k by 2n multiplication table. If a coset is assigned the number C, then the entries in row C of the table are the numbers corresponding to Cg_i and Cg_i^{-1} for each generator of G. For example, the complete coset table of $H = \langle b^{-1} \rangle$ in $G = \langle a, b \mid a^4 = (ab)^2 = b^2 = 1 \rangle$ is in figure 2.

Information for a coset table is obtained by forming subgroup and relation tables. Let h be a generator of H. Then h is a word in the generators of G. Suppose $h = x_1x_2...x_t$. A blank subgroup table for h is in figure 2. The entry in the column headed by x_i is the number of the coset $H(x_1x_2...x_i)$. Since $x_1x_2...x_t$ is a generator of H, it is an element of H. Hence $H(x_1x_2...x_t) = H$, and the entry in the last column of the table will always be 1.

Let R be a relation of G. Then $R = y_1y_2...y_s$, a word in the generators of G. A relation table is similar to a subgroup table, but it contains a row for each coset number. A blank relation table for R is in figure 2. In such a relation table, the entry in the j^{th} row and the column headed by x_i is the coset number assigned to the coset $jy_1y_2...y_i$. Note that $jy_1y_2...y_s = j(1) = j$. Hence the last entry in the j^{th} row will always be j. For the remainder of this chapter, when referring to an arbitrary subgroup or relation table, we will assume that it is formed by the word $x_1x_2...x_t$, and the column of a table headed by x_i will be denoted as the

(a) The coset table for $H = \langle b^{-1} \rangle$ in $G = \langle a, b \mid a^4 = (ab)^2 = b^2 = 1 \rangle$

	a	b	a^{-1}	b^{-1}
1	2	1	4	1
2	3	4	1	4
3	4	3	2	3
4	1	2	3	2

(b) A blank subgroup table for the subgroup generator $x_1x_2 \ldots x_t$

	x_1	x_2	...	x_t
1				

(c) A blank relation table for $R = y_1y_2 \ldots y_s = 1$

	y_1	y_2	...	y_s
1				
2				
⋮				
k				

Figure 2. Examples of coset, subgroup, and relation tables.

i^{th} column, neglecting the first column of coset numbers. Let c_i and c_{i+1} be the entries in columns i and i+1 of a subgroup table or a row of a relation table. Then clearly c_{i+1} is the number of the coset $c_i x_{i+1}$. Also c_i is the coset number assigned to $c_{i+1}x_{i+1}^{-1}$. Hence each element in the tables reveals information for the coset table. In fact, it can be shown [Tr] that if [G:H] is finite, then the subgroup tables and relation tables contain all of the information necessary to complete the coset table. As an example, the complete subgroup and relation tables for $G = \langle a, b \mid a^4 = (ab)^2 = b^2 = 1\rangle$ and $H = \langle b^{-1}\rangle$ are in figure 3.

The idea behind the TC algorithm is to construct the subgroup tables and then the relation tables row by row, adding new information to the coset table during the process. When constructing a subgroup table or a row of a relation table, one uses any information already stored in the coset table. Since the last column is always known, one can fill in the rows from both directions. If at any point it becomes impossible to fill in a row any further using the information already in the coset table, then a new coset must be defined as the next available integer. At the point where a row fills in, there are three possibilities. If information for one or both of the last two cosets entered into the row is empty in the coset table, then new multiplication can be defined in the coset table. Multiplication defined in this manner is called a deduction. If there is already information in the table for either of the last two entries, then two things can happen. The information obtained from the row may be redundant, requiring no further action. Perhaps the new information from the row will disagree with the information in the table. This occurs when two numbers have been assigned to the same coset. This is called a coincidence. The convention for handling coincidences is to replace the higher number with the smaller. Several authors, including Trotter [Tr], have

(a) Subgroup Table

	1/b
1	1

(b) Relation Table for $b^2 = 1$

	b	b
1	1	1
2	4	2
3	3	3
4	2	4

(c) Relation Table for $a^4 = 1$

	a	a	a	a
1	2	3	4	1
2	3	4	1	2
3	4	1	2	3
4	1	2	3	4

(d) Relation Table for $(ab)^2 = 1$

	a	b	a	b
1	2	4	1	1
2	3	3	4	2
3	4	2	3	3
4	1	1	2	4

Figure 3. The subgroup and relation tables for $G = \langle a, b \mid a^4 = (ab)^2 = b^2 = 1\rangle$ and $H = \langle b^{-1}\rangle$.

shown that proceeding through the rows of the subgroup and relation tables handling definitions, deductions, and coincidences will terminate, filling the coset table, as long as [G:H] is finite. There are many ways to implement the TC algorithm. Some methods attempt to conserve storage space. Others attempt to speed up computations. The reader is encouraged to read the articles [CD], [Le1], [Le2], [Ne], and [Tr] listed in the bibliography to see explanations of various implementations.

The version implemented by Maple also creates a list of coset representatives for H in G. The first coset representative is the identity, representing H. If a coset c times a generator g is defined to be a new coset k, then the coset representative for k can be defined as the concatenation of the coset representative for c with g. Maple does this during the process to store the coset representatives as a list called reps.

MAPLE'S IMPLEMENTATION OF THE BASIC TC ALGORITHM

Maple has an internal routine `group/tc` that is called by the other commands for finitely presented groups. This procedure returns a list of coset numbers, coset representatives, and the augmented coset table. Maple handles the subgroup tables first, then it handles each row of each relation table one at a time. Maple never actually stores the subgroup or relation tables. An internal routine `group/construct` works through the information in the row being examined and stores new information into the coset table. It uses the fact that each row begins and ends with the same coset number to work in two directions. Initially the first and last entries are stored as the variables lval and rval respectively. The first and last generators are stored as lgen and rgen. Then Maple attempts to construct the element to the immediate left of lval as lnext =lval*lgen and the element immediately to the right of rval as rnext=rval*1/rgen. If lnext or rnext are defined in the coset table, then that entry replaces lval or rval respectively, lgen and rgen are updated, and lnext or rnext is calculated again. If neither entry is available in the coset table, then lval is defined to be the next available integer n not already representing a coset. Then lval*lgen = n and n*(1/lgen) = lval are stored in the coset table, and the n^{th} coset representative is stored as the coset representative of lval times lgen. This is repeated until Maple has entered values for all elements in the given row. Then Maple has to consider the three cases previously mentioned to find deductions or coincidences.

For example consider $G = \langle a, b \mid baba^{-1}a^{-1} = abab^{-1}b^{-1} = 1\rangle$ and $H = \langle a^2\rangle$. We will show in detail how Maple uses the subgroup table to obtain information. First lval and rval are assigned coset number 1. Then Maple attempts to form lnext and rnext. Both entries are empty in the coset table. Hence lnext is assigned the number 2 and 1a is stored as 2 in the coset table. Also $2a^{-1}$ is stored as 1. Reps[1] has already been defined as [1] by the main procedure `group/tc.` Thus Reps[2] can now be stored as [1&*a]. The symbol &* is a neutral operator used by Maple to form concatenation of words. The neutral symbols can be removed at the end of the algorithms with the command `group/flatten.` The definition of 1a = 2 completes the table and leads to the deduction 2a = 1 and $1a^{-1} = 2$, which are stored in the coset table. Now Maple has some information to use for future calculations.

The procedure construct is called repeatedly to construct the information in the relation tables. We will describe in detail the construction of the first row of the table for $baba^{-1}a^{-1}$ and the construction that involves the only occurrence of a coincidence in this example. The data obtained by the constructions is in figure 4. To construct the first row of the relation table for $baba^{-1}a^{-1}$, Maple assigns lval and rval both 1's representing the first coset. Then Maple attempts to form lnext = 1b and rnext = 1a. The value of 1b is undefined, but 1a = 2. Hence, rval is assigned 2, and Maple attempts to find rnext = 2a. This is defined already as 1. So rval is assigned 1 and Maple attempts to find rnext = $1b^{-1}$ which is undefined. Thus, a new coset must be defined. Maple defines 1b = 3, which allows 1b = 3 and $3b^{-1}$ = 1 to be entered into the table. Maple stores 1*b as the third coset representative. Then lval gets assigned 3 and Maple attempts to find lnext = 3a which is undefined. Hence, Maple defines 3a = 4 and stores this and $4a^{-1}$ = 3 into the coset table and sets reps[4]= [(1&*b)&*a]. This completes the row and defines 4b = 1 and $1b^{-1}$ = 4 which are stored in the coset table.

Now suppose that Maple has already called construct eight times and is going to perform the ninth construction. This will be the fifth row of the table for the relation $baba^{-1}a^{-1}$. First lval and rval are set equal to 5. Then lnext and rnext are calculated as 4 and 6 respectively. These are stored as lval and rval, and Maple calculates lnext = 8 and rnext = 7. This completes the row giving 5 | 4| 8| 7| 6| 5. This creates the deduction 8b =7. However, 8b has already been defined to be 9. This is because 7 and 9 are actually representing the same coset. Hence Maple has encountered a coincidence. When this occurs, one of the numbers must be replaced by the other. Since it is desired that the cosets to be numbered 1 to [G:H], the smaller number should replace the larger. If one were calculating the tables by hand, it would be easiest to simply replace every occurrence of a 9 with a 7. This is not as easy to deal with during the Maple procedure. Hence, Maple uses the procedures substitute and alias to handle coincidences. Generic outlines of these algorithms are listed below.

routine: substitute
input: coset table, subtable, two numbers representing the same coset
output: modified coset table and subtable

1. Find the aliases (as described below) for the two integers
2. Enter the smaller number into the subtable entry for the larger.
3. Fill in the coset table for the smaller number with any additional information available in the table from the larger number, which may lead to additional coincidences.
4. Eliminate the entries for the larger integer from the coset table and the list of coset representatives
5. Repeat for any additional coincidences discovered in step 3.

routine: alias
input: an integer c representing a coset

(a) Subgroup Table

	a	a
1	2	1

(b) Relation table for $baba^{-2} = 1$

	b	a	b	1/a	1/a
1	3	4	1	2	1
2	5	6	2	1	2
3	6	7	8	3	3
4	1	2	5	8	4
5	4	8	7	6	5
6	2	1	3	7	6
7	8	5	4	3	2
8	7	3	6	5	8

(c) Relation Table for $abab^{-2} = 1$

	a	b	a	1/b	1/b
1	2	5	6	3	1
2	1	3	4	5	2
3	4	1	2	6	3
4	8	9	3	1	4
5	6	2	1	4	5
6	7	8	5	2	6
7	3	6	7	8	7
8	5	4	8	7	8

(d) Partial coset table before coincidence

	a	b	1/a	1/b
1	2	3	2	4
2	1	5	1	6
3	4	6	9	1
4	8	1	3	5
5	6	4	8	2
6	7	2	5	3
7		8	6	
8	5	9	4	7
9	3			8

(e) Coset table after coincidence

	a	b	1/a	1/b
1	2	3	2	4
2	1	5	1	6
3	4	6	9	1
4	8	1	3	5
5	6	4	8	2
6	7	2	5	3
7	3	8	6	8
8	5	9	4	7

Figure 4. Subgroup, Relation, and Coset Tables for $G = \langle a, b \mid baba^{-1}a^{-1} = abab^{-1}b^{-1} = 1 \rangle$ and $H = \langle a^2 \rangle$.

output: the smallest integer representing that coset

1. set ans = c
2. while subtab[ans] is defined do
   ans:= subtab[ans]
   enddo
3. return ans

Running the substitute procedure transfers any information from the larger coset integer to the smaller and removes the entries for the larger integer from the coset table and representatives. It does not remove any occurrences of the larger integer from other slots in the table. That must be done at the end by recalculating the table entries with the routine `group/mulcoset.` This procedure performs multiplication of cosets by generators using the existing table, then taking the alias of the result.

MODIFIED TC ALGORITHM

The above is an outline of how Maple performs the Todd-Coxeter algorithm to find coset representatives and a coset table for a subgroup of a finitely presented group. Maple actually performs a modified version of the TC algorithm to allow for any element of a finitely presented group to be defined as the product of an element in a subgroup and a coset representative. Instead of just storing the numbers of the cosets in the table, it stores pairs. If cg = k for some coset numbers c and k and generator g, then a basic TC coset table stores a k. In an augmented table formed by the modified TC algorithm, a pair [h,k] is stored where $h(\bar{k}) = (\bar{c})g$ with $\bar{k}$ and $\bar{c}$ the coset representatives of c and k respectively.

The augmented coset table is constructed using augmented subgroup and relation tables. For any coset number c_i, denote the coset representative as $\bar{c}_i$. If the table is formed by the word $x_1x_2...x_n$, then the i^{th} entry in a row of the table is a pair $[h_i,c_i]$, where $h_i\,\bar{c}_i = \bar{c}_0x_1x_2...x_n$ with c_0 the coset number starting the row. Note that for relation tables, the first and last entries may be defined as $[1, c_0]$. If $x_1...x_n$ is a generator of the subgroup, then we may define [1, 1] as the first element in the subgroup table, and the last element in the table may be defined as $[x_1...x_n, 1]$ since $x_1...x_n\bar{1} = \bar{1}x_1...x_n$. To simplify calculations and allow the procedure pres to work, which will be discussed shortly, a subgroup generator $x_1...x_n$ must be entered into Maple as $y = x_1...x_n$ where y is any user defined name. Thus occurrences of $x_1...x_n$ in the table can be replaced by y, creating shorter tables. Recall that there are three ways to obtain information for the coset table. These are by definition, deduction, or coincidence. Suppose that the TC algorithm defines $c_ix_{i+1} = c_{i+1}$. Then Maple stores $\bar{c}_i$ multiplied by x_{i+1} as the coset rep for c_{i+1}. Hence, the augmented coset table entry for $(c_i)(x_{i+1})$ is $[1, c_{i+1}]$, and the entry for $(c_{i+1})(x_{i+1}^{-1})$ is $[1,c_i]$.

Recall that construct fills in rows using mulcoset and by defining new cosets. Suppose that lval = $[h_i, c_i]$. If $(c_i)(x_{i+1})$ is defined in the coset table as $[h_{i+1},c_{i+1}]$, then mulcoset returns $[h_i\&*h_{i+1},c_{i+1}]$. Note that $h_ih_{i+1}\bar{c}_{i+1} = h_i\bar{c}_ix_{i+1} = \bar{c}_0x_1x_2...x_{i+1}$. Analogous results are true for rval with mulcoset. If $(c_i)(x_{i+1})$ is not defined in the coset table, then it is defined as previously stated, and lval is replaced by $[h_i, c_{i+1}]$ where $c_{i+1} = c_ix_{i+1}$.

Deductions occur when a row closes and lval becomes adjacent to rval. Suppose that the table closes with lval in the n^{th} entry of the row. Then lval = $[h_n, c_n]$ where $h_n\bar{c}_n = \bar{c}_0x_1...x_n$, and rval = [g, d] with $g\bar{d} = \bar{c}_0x_1...x_nx_{n+1}$. If lnext is undefined, Maple makes the deduction $(c_n)(x_{n+1})$ = [1/lval[1]&*rval[1], rval[2]] where rval[i] and lval[i] are the i^{th} components of the ordered pairs. Note that the correctness of the table requires that there exist h_{n+1} in H such that $h_{n+1}c_{n+1} = c_nx_{n+1}$. So 1/lval[1]*rval[1]*$\overline{\text{rval[2]}}$ = $(h_n^{-1})(g)(\bar{d}) = (h_n^{-1})\ \bar{c}_0x_1...x_nx_{n+1} = (h_n^{-1})\ h_n\bar{c}_nx_{n+1} = h_{n+1}\bar{c}_{n+1}$ which is precisely the desired element. Similarly, if rnext is undefined, then $(d)(x_{n+1}^{-1})$ can be defined as [1/rval[1]&*lval[1], lval[2]].

This leaves only coincidences to consider. A coincidence occurs if the table fills in and rnext[2] is not equal to lval[2] or lnext[2] is not equal to rval[2]. In this occurrence, there are two pairs $[k_1,j_1]$ and $[k_2,j_2]$ that are equivalent, and we need to replace the larger coset number with the smaller one. This is handled with substitute and alias. If $j_1<j_2$, substitute places $[1/k_2\&*k_1, j_1]$ into the subtable for j_2 for use with alias. Alias uses the subtable to replace $[k_2, j_2]$ with $[k_2\&*1/k_2\&*k_1, j_1]$. These procedures are actually defined recursively to insure that the smallest coset number is used if more than two numbers represent the same coset. Mulcoset uses alias to ensure that the smallest coset number is used in all future calculations.

Note that the subgroup elements stored are very nasty nested concatenations of words in the renamed generators of H. These can be simplified with the command `group/flatten` which removes any parenthesis, ones, or &*'s. If the algorithm requires the formation of a large amount of coset numbers, then the concatenations can become too large to handle. If this occurs Maple will return an error.

MAPLE'S PROCEDURES FOR FINITELY PRESENTED GROUPS

Setting up Groups and Subgroups

Similar to permgroup, grelgroup can be used to set up a finitely presented group to be used by other commands. If $G = \langle g_1, g_2, ..., g_n \mid R_1 = R_2 =... = R_m = 1\rangle$, then G can be entered into Maple as G:= grelgroup($\{g_1, g_2, ..., g_n\}$, $\{R_1, R_2, ..., R_m\}$) where each R_i is a list representing a word. For example, $G = \langle a, b \mid a^4 = b^2 = (ab)^2 = 1\rangle$ can be setup with the command G:=grelgroup({a, b},{[a, a, a, a], [b, b], [a, b, a, b]}). The routine checks to ensure that the arguments are of the correct type and that each R_i contains only generators of G or their inverses. If everything checks out appropriately, then the group is set up as a function to be called by other commands.

The command subgrel can be used to set up a subgroup generated by words in the generators of G. For instance, if G is set up as above, then H = $\langle ab^{-1} \rangle$ can be defined by the command H:=subgrel({h = [a, 1/b]}, G). Note that each generator must be entered as a name and a word in the generators of G. These are used by `group/tc` to construct the augmented coset table with words in the renamed generators of H. Maple checks to ensure that the given generators are correctly defined and sets up a function to be called by other commands.

Cosets

The method that Maple uses to construct a coset table and a set of coset representatives for a subgroup H of a finitely presented group G has already been discussed. The routine `group/tc` returns a list of coset numbers, coset representatives, and a coset table. The command cosets(H) calls `group/tc` and returns the set of coset representatives. Note that if G is finite with relations $\{R_1 = R_2 = ... = R_m = 1\}$, then one can define the trivial subgroup H:= subgrel({h=R_1}, G). Then cosets(H) will list the elements of G. For example suppose that G:=grelgroup({a, b},{[a, a, a, a], [a, b, a, b], [b, b]}) and H:= subgrel({h=[b, b]},G). Then cosets(H) returns the set *{[], [b], [a], [a, a], [a, b], [b, a], [a, a, a], [b, a, a]}*.

The Procedure cosrep

Once the augmented coset table has been formed, it is easy to write any element of G as a product of an element of H and a coset representative. The internal routine `group/tc` returns three things. These are the coset numbers, the coset representatives, and the augmented coset table. Using these, the following algorithm can be used to calculate the desired subgroup element and coset representative.

routine: cosrep
input: an element g of G expressed as a word in the generators and a subgroup H
output: [h,k] such that h is an element of H and k is a coset representative for H in G and hk = g
1. Set mt:=coset table, cr:=coset representatives, cn:= 1, and sgw:=1
2. for each generator in g do
    dc:= mt[cn][gen]
    sgw:=sgw&*dc[1]
    cn:=dc[2]
  enddo
3. return(`group/flatten`(sgw), cr[cn])

Suppose that G:= grelgroup({a, b},{[a, b, a, 1/b, 1/b], [b, a, b, 1/a, 1/a]}) and H:= subgrel({h=[b]},G). Then cosrep([1/a], H) returns *[[1/h,1/ h], [a,b]]*, meaning that 1/a = $(1/h)^2(ab)$ = $(1/b)^2(ab)$. The command cosrep([b, b], H) returns *[[h, h], []]*, meaning that $b^2 = h^2(e_G)$. This shows that $b^2 \in$ H, which is obvious in this case.

Representing a Group as a Permutation Group

Cayley's theorem states that any group is isomorphic to a group of permutations. Gallian proves Cayley's theorem [Ga1 p. 119-120] by showing that a group G is isomorphic to $\{T_g \mid g \in G\}$ where $T_g(x) = gx$ for all x in G. Each $T_g(x)$ is a permutation on the elements of G. Clearly such a group is generated by elements of the form $T_{g'}$ where g' is a generator of G. This isomorphism can be weakened to a homomorphism if cosets are used instead of group elements. Thus to find a permutation group that is a homomorphic image of G, we need only to find generators T'_g such that $T'_g(Ha) = Hag$ for each generator g of G. These are precisely the columns of a coset table for H in G. The procedure permrep uses `group/tc` to construct the augmented coset table. Then it forms a set of generators for a permutation group by extracting the columns of the coset table. If G is finite, one can use the full power of Cayley's theorem to find a permutation representation for G by defining H to be the trivial subgroup. For instance, suppose that G:= grelgroup({a, b},{[a, b, a, 1/b, 1/b], [b, a, b, 1/a, 1/a]}). Then the trivial subgroup can be defined as H:= subgrel({h=[a, b, a, 1/b, 1/b]},G). Then a permutation representation for G can be found by typing permrep(H) which returns *permgroup(24,{a = [[1, 2, 7, 10, 11, 19], [3, 4, 13, 16, 22, 24], [5, 6, 14, 15, 18, 20], [8, 9, 12, 21, 23, 17]], b = [[1, 5, 4, 10, 15, 22], [2, 3, 9, 11, 16, 23], [6, 7, 8, 18, 19, 21], [12, 13, 14, 17, 24, 20]]}).*

Random Elements

It is not hard to construct an arbitrary element of a group defined by a set of generators and relations. One simply has to select a random set of generators or their inverses and multiply the elements of the set together. Suppose that G is generated by n elements $\{g_1, g_2, ..., g_n\}$. Then the procedure RandElement(G) selects six random numbers between 1 and 2n. If the random number is greater than n then Maple selects $1/(g_{r-n})$. Otherwise, it selects g_r. Then these six elements are multiplied together using `group/mulword.` The routine `group/mulword` performs concatenation on the words, eliminating any occurrences of generators that appear beside their inverses. If the word reduces to [], then Maple repeats the procedure. If G:= grelgroup({a, b}, {[a, b, a, 1/b, 1/b], [b, a, b, 1/a, 1/a]}), then RandElement(G) will return a random word in a, b, 1/a, and 1/b such as [a, a, 1/b, 1/a]. This particular group can be shown to have 24 elements which can be viewed using cosets as previously mentioned. The elements produced by RandElement may appear different than those 24 elements since there is no attempt to reduce words in the generators.

Testing Normality

Recall that a subgroup H is normal in G if and only if $ghg^{-1} \in H$ for all generators g of G and all generators h of H. Group membership for a subgroup of a finitely presented group can be tested using cosrep. For each g in G the command cosrep(g, H) returns a pair of elements [h, k] such that k is a coset number, h is in H, and $h\bar{k} = g$. Clearly, if $g \in H$, then $\bar{k}$ must be []. The procedure isnormal uses this fact to

determine whether or not $ghg^{-1} \in H$ for all generators g of G and all generators h of H. If G:= grelgroup({a, b}, {[a, b, a, 1/b, 1/b], [b, a, b, 1/a, 1/a]}) and H:= subgrel({h1=[a, b], h2=[b, a]}, G), then isnormal(H) returns *true*. If H:= subgrel({h1=[a,b],h2=[b,a,b,a]},G), then isnormal(H) returns *false*.

Finding a Finite Presentation for a Subgroup

The Todd-Coxeter algorithm calculates the index of a subgroup H of a finitely presented group G. It does not however provide a set of defining relations for H. Much research has gone into calculating a presentation for a subgroup of a finitely presented group. Several methods exist for tackling this problem. One method called the Reidemeister-Schreier method [GAP2] uses Schreier generators and techniques called Reidemeister rewriting and Tietze transformations. A version of this method is included in the software GAP. Anyone desiring more details should consult the GAP web page at St. Andrews University [GAP1]. Other techniques are based on the modified Todd-Coxeter algorithm.

D.H. McLain [Mc] describes a simple technique for calculating a set of defining relations for a subgroup of a finitely presented group. If one has constructed an augmented coset table, as Maple does, then it can be used to rewrite $\bar{c}\,R$ for each coset c and each relation R as the product of a word in the generators of H and a coset representative, using an algorithm identical to cosrep. Suppose $\bar{c}\,R = h_1h_2...h_s\bar{k}$. Since R = 1, $h_1h_2...h_s\bar{k} = \bar{c}$. Thus $\bar{k} = \bar{c}$, and $h_1h_2...h_s = 1$. This gives a relation for H. McLain proves that if each generator of H is a generator of G, then a complete set of relations for H can be found by performing this rewriting for each coset and relation of G. McLain quotes an example of Mendelsohn [Me] that shows that these relations are not enough to define H if the generators of H are not generators of G. Mendelsohn notes that for the group $G = \langle a, x | x^{-1}a^2x = a^3\rangle$ with subgroup $H = \langle x, a^8\rangle$, there is one relation and coset representative. Hence, McLain's procedure can only form one relation for H, but as Mendelsohn points out, G. Higman has shown that H, which is equal to G, requires two relations in terms of x and a^8. Thus additional relations are required.

Neubuser, in [Ne] shows exactly what extra relations must be added to those of McLain's in order to define a subgroup H of a group G. Suppose that $g_1g_2...g_t$ is a generator of H expressed as a word in the generators of G. Suppose that the augmented coset table reveals that $\bar{1}\,g_1g_2...g_t = g_1g_2...g_t = h_1h_2...h_s\bar{k}$ where $h_1h_2...h_s$ is a word in the generators of H and $\bar{k}$ is the coset representative of the k^{th} coset. Then clearly $\bar{k} = \bar{1}$, and $g_1g_2...g_t = h_1h_2...h_s$. This gives a relation for H. Neubuser proves that the set of relations formed by this procedure for each generator of H and the relations formed by McLain's technique are enough to define H.

Maple's procedure pres calculates a presentation for a subgroup of a finitely presented group generated by a set of words in the generators of G. Maple's procedure uses a clever method to calculate all of the relations proven necessary by Neubuser with a process as simple as McLain's. Instead of simply defining the generators of H as words in the generators of G, the generators are given a name which creates

a relation. For example, if $G = \langle a, x | x^{-1}a^2x = a^3 \rangle$ and $H = \langle x, a^8 \rangle$, Maple requires that H be entered as subgrel({h1 = [x], h2 = [a, a, a, a, a, a, a, a]},G) after G has been properly entered. Of course, h1 and h2 can be any names assigned by the user. The relations h1 = [x] and h2 = [a, a, a, a, a, a, a, a] are essentially those described by Neubuser since each subgroup table is built such that the last entry is [y,1] where y is the name assigned to the generator of H. The augmented coset table for this example is built with h1 and h2 instead of x and a^8. Thus the relations formed by rewriting the relations of G will contain only h1, h2, and their inverses. Hence by simply renaming the generators of H, the procedure pres can use McLain's technique to form a defining set of relations for a subgroup. If one sets G:= grelgroup({a, x},{ [1/x, a, a, x, 1/a, 1/a, 1/a] }) and H:=subgrel({h1=[x], h2=[a, a, a, a, a, a, a, a]},G) then pres(H) returns the complicated presentation *grelgroup({h1, h2},{[1/h1, h2, 1/h1, 1/h2, h1, h2, 1/h1, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h1, h2, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h2, 1/h1, 1/h2, h1, h2, h2, 1/h1, 1/h2, h1, h2, 1/h1, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h1, h2, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h2, 1/h1, 1/h2, h1, h2, 1/h1, 1/h2, h1, h2, h1, 1/h2, 1/h1, h2, h1, 1/h2, 1/h1, h2, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h1, h2, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h1, h2, 1/h1, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h1, h2, 1/h1, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h1, h2, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h2, 1/h1, 1/h2, h1, h1, h2, 1/h1, 1/h2, 1/h1, h2, h1, 1/h2, h1, h2, 1/h1, 1/h2, h1, h2]}).* At first, it appears that this is an incorrect presentation since there is only one relation. Note, however, that Mendelsohn claims in [Me] that there must be two relations to define G in terms of x and a^8. This presentation is in terms of h1 and h2, and the second relation has been embedded in the definition of h2.

Finding the Size of a Group

The procedure grouporder defines a group to be infinite if it has more generators than relations which is proven in proposition 4.1. If this is not the case, then the procedure calculates a random element g, and forms the presentation of $\langle g \rangle$ as described above. Clearly if $\langle g \rangle$ is infinite, the relations should reduce to []. Otherwise, g would satisfy the relation and have finite order. If g has finite order then it is calculated as the gcd of the lengths of the relations. Note that each relation is a list of g's or a list of g^{-1}'s since those are the only possibilities. Thus each generator is of the form $g^n = 1$ for some n or $g^{-n} = 1$. Note $g^{-n} = 1$ if and only if $g^n = 1$. It a result of elementary group theory that $g^n = g^m = 1$ if and only if $g^{gcd(m,n)} = 1$. Hence the later can replace both of the former as a single relation. Thus Maple's method is valid for calculating the order of a cyclic subgroup. After the order of H is found, the order of G is calculated as |H|[G:H] using LaGrange's theorem. If G:=grelgroup({a,b},{[a,b,a,1/b,1/b],[b,a,b,1/a,1/a]}), then grouporder(G) returns *24*.

Proposition 4.1: Let $G = \langle g_1, g_2, ..., g_n \mid R_1 = R_2 = ... = R_m = 1 \rangle$ with n > m. Then G is infinite.

Proof: Consider the group $A = \langle g_1, g_2, ..., g_n \mid R_1 = R_2 = ... = R_m = [g_i, g_j] = 1 \text{ for } 1 \leq i, j \leq n \rangle$ where $[a, b] = aba^{-1}b^{-1}$. The additional relations ensure that A is an Abelian group. By Van Dyck's Theorem [Hu p. 67], A

is an epimorphic image of G. Hence, if A is an infinite group, then G is an infinite group. To show that A is infinite, we will use results from [Cai]. By theorem 6 [Cai p. 228] and its proof, A is isomorphic to F/N where F is the free Abelian group on $\{g_1, g_2, ..., g_n\}$ and $N = \langle R_1, R_2, ..., R_m \rangle$ in F. Since F is a free Abelian group generated by $\{g_1, g_2, ..., g_n\}$, there exists e_{ij} such that

$$R_i = \sum_{j=1}^{n} e_{ij} g_j \qquad (i = 1, ..., m).$$

This gives an m by n matrix

$$M = \begin{bmatrix} e_{11} & e_{12} & \cdots & e_{1n} \\ e_{21} & e_{22} & \cdots & e_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ e_{m1} & e_{m2} & \cdots & e_{mn} \end{bmatrix}.$$

Cairns defines two matrices to be equivalent if and only if each of them is the matrix of a set of equations like those above for the R_i's for some set of m generators of N in terms of a basis for F. He then proves that if γ is the rank of M, then M is "equivalent to an $m \times n$ matrix whose elements are all zeros except for the first γ elements on the main diagonal, which are positive integers $(\tau_1, \tau_2, \ldots, \tau_\gamma)$ such that τ_{i+1} divides τ_i (i = 1, ..., γ -1)" [Cai p. 224]. This equivalent matrix gives the structure of A. By theorem 7 [Cai p. 228], A is isomorphic to

$$Z_{\tau_1} \oplus Z_{\tau_2} \oplus \cdots \oplus Z_{\tau_\gamma} \oplus Z \oplus Z \oplus \cdots \oplus Z$$

where there are $n - \gamma$ copies of Z. For this particular group A, $n - \gamma \geq n - m > 0$. Therefore, A is infinite. Hence, G is infinite. ■

CONCLUSION

The procedures in the chapter, driven by the Todd-Coxeter algorithm, can be used obtain a great deal of information about groups given by very abstract definitions. The TC algorithm only requires that [G:H] be finite. Thus results can be obtained for some infinite groups, unlike the procedures in chapter 2. Many adaptations and improvements have been made since Todd and Coxeter first published their results in 1936. However, much remains unknown about the algorithm. Different presentations of the same group can behave very differently. The order in which the relation tables are constructed can greatly affect the algorithm. There are presentations of the trivial group that have been shown to require the formation of millions of cosets before proving that the group is trivial. These questions and others are providing researchers with ample material to continue work in computational group theory. Wide access to computers did not even become available until the last half of the twentieth century, so this is still a very young branch of mathematics.

CHAPTER 5: APPLICATIONS OF GROUP THEORY

In addition to its theoretical significance, group theory has been shown to have applications in many fields including physics and chemistry. In this final chapter, the author will discuss three possible applications. These are check digit schemes based on dihedral groups, RSA encryption, and solving puzzles using permutations. These are just a few simple examples to show that group theory can be a powerful tool. All three examples take advantage of both group theory and computing to solve real world problems.

CHECK DIGIT SCHEMES BASED ON D_n

Whenever information is being transferred by humans or by machines, errors are likely to occur. This is especially true when the information is a string of digits or letters. Perhaps a digit will get left out, or an extra digit will be inserted. Perhaps two adjacent digits will get switched. It would be nice if one could somehow check a string of digits to see if any errors have occurred in transmission. One way to do this is to append a check digit onto the end of the string. This digit and the digits of the string should satisfy some mathematical equation. Several check digit schemes have been developed. For a comprehensive survey of these methods consult [Ga2]. The author will discuss a couple of methods using the dihedral groups.

A single check digit scheme based on D_5

Schemes involving the dihedral groups have been shown to detect more errors than schemes based on modular groups. The author has written Maple procedures to implement a scheme based on D_5 described in [Ga1]. The method devised by J. Verhoeff in 1969 has been shown to detect "all-single digit errors and all transposition errors involving adjacent digits without the necessity of avoiding certain numbers or introducing a new character" [Ga1 p.104-105]. To work with the digits 0 through 9, the elements of D_5 can be renamed with these digits. The complete multiplication table for D_5 in terms of the digits 0 through 9 is given in table 1.

Consider the permutation s = (01589427)(36). Verhoeff's idea is to append a check digit a_n onto the string of digits $a_1a_2...a_{n-1}$, so that $s(a_1)* s^2(a_2) *...* s^{n-2}(a_{n-2}) * s^{n-1}(a_{n-1}) * s^n(a_n) = 0$ where $s^n(x) = s(s^{n-1}(x))$. The action of s on D_5 is represented in table 2. The table shows that $s^i(a) = s^i(b)$ if and only if a = b. Thus the check digit will detect all single-digit errors. The table also shows that a*s(b) = b*s(a) if and only if a = b. Hence $s^i(a)*s^{i+1}(b) = s^i(b)*s^{i+1}(a)$ if and only if a =b. Therefore the scheme will detect all transposition errors involving adjacent digits.

As Gallian states, "In 1990 the German government began using a minor modification of Verhoeff's check digit scheme to append a check digit to the serial numbers on German banknotes" [Ga1 p. 105]. The German banknote serial numbers consist of alphanumeric strings of length ten. The letters are assigned numbers according to table 3. Instead of appending a_{11} to the end of the string such that

Table 1. Multiplication in D_5

*	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	0	6	7	8	9	5
2	2	3	4	0	1	7	8	9	5	6
3	3	4	0	1	2	8	9	5	6	7
4	4	0	1	2	3	9	5	6	7	8
5	5	9	8	7	6	0	4	3	2	1
6	6	5	9	8	7	1	0	4	3	2
7	7	6	5	9	8	2	1	0	4	3
8	8	7	6	5	9	3	2	1	0	4
9	9	8	7	6	5	4	3	2	1	0

Table 2. The action of s^i on D_5

	0	1	2	3	4	5	6	7	8	9
s	1	5	7	6	2	8	3	0	9	4
s^2	5	8	0	3	7	9	6	1	4	2
s^3	8	9	1	6	0	4	3	5	2	7
s^4	9	4	5	3	1	2	6	8	7	0
s^5	4	2	8	6	5	7	3	9	0	1
s^6	2	7	9	3	8	0	6	4	1	5
s^7	7	0	4	6	9	1	3	2	5	8
s^8	0	1	2	3	4	5	6	7	8	9
s^9	1	5	7	6	2	8	3	0	9	4
s^{10}	5	8	0	3	7	9	6	1	4	2

Table 3. Banknote numeric assignments

A	D	G	K	L	N	S	U	Y	Z
0	1	2	3	4	5	6	7	8	9

$s(a_1)*s^2(a_2)*...*s^{10}(a_{10})*s^{11}(a_{11}) = 0$, the German's append a digit a_{11} such that the equality $s(a_1)*s^2(a_2)*...*s^{10}(a_{10})*a_{11} = 0$ holds. As Gallian points out, the German scheme will not detect the switching of a letter with its integer representative, and it will not detect all of the transpositions of adjacent digits involving the check digit itself, such as $a_1a_2...a_9\ a_{11}a_{10}$.

The author has implemented the German check digit scheme for ten digit numerical strings in Maple. The multiplication table for D_5 is stored as a 10 by 10 matrix D5. The procedure md5 uses this table to perform multiplication. Similarly $s^i(x)$ is calculated by the procedure sig using a matrix ST, containing the information in table 2. The procedure FormCheckDigit finds the appropriate check digit for a ten digit string entered as a list. For example FormCheckDigit([0, 2, 8, 5, 3, 6, 8, 2, 7, 7]) returns the check digit 7. The procedure Check checks an 11 digit list for errors using the same equation as the German banknote scheme. Check([0, 2, 8, 5, 3, 6, 8, 2, 7, 7, 7]) returns *correct.* Check([0, 2, 8, 3, 5, 6, 8, 2, 7, 7, 7]) returns the message *There is an error*. Thus, the scheme correctly detects the transposition error 5, 3 → 3, 5. The command Check([0, 2, 9, 5, 3, 6, 8, 2, 7, 7]) returns *There is an error*, detecting the replacement of 8 with a 9. Hence, the scheme is effective at detecting simple errors.

A double check digit scheme

As mentioned the above check digit scheme does not pick up all mistakes. A more powerful scheme using two check digits is described by Steven Winters [Wi]. As Winter's shows, his scheme detects all of the following errors:

1. Single errors a → b ($a \neq b$)
2. Transposition errors ab→ba ($a \neq b$)
3. Twin errors aa→bb ($a \neq b$)
4. Double adjacent errors ab→ cd ($a \neq c, b \neq d$)
5. Triplet errors aaa→bbb ($a \neq b$)
6. Triple consecutive errors abc → def ($a \neq d$, $b \neq e$, $c \neq f$)
7. Jump transposition errors abc → cba ($a \neq c$)
8. Double transposition errors abcd → cdab ($a \neq c$, $b \neq d$)
9. Insertion errors
10. Deletion errors

Winters scheme works with any D_n when n is an odd integer using the permutation s = (0)(1, n-1)(2, n-2)...((n-1) / 2, (n+1) / 2)(n, n+1, ..., 2n-1). Instead of appending a single check digit to a string $...a_{2i+2}a_{2i+1}a_{2i}a_{2i-1}...a_4a_3a_2a_1$, Winters appends two digits z_1 and z_2 such that $...* s^{i+1}(x_{2i+2}) * s^i(x_{2i}) *...* s^2(x_4) * s(x_2) * z_2 = 0$ and $...* s^{i-1}(x_{2i+1}) * s^i(x_{2i-1}) *...* s^2(x_3) * s(x_1) * z_1 = 0$.

The author has implemented Winter's scheme using D_5 to append two check digit onto a ten digit string. The procedure checkdigits returns two check digits for a list of ten digits. For example checkdigits([1, 2, 3, 4, 5, 3, 2, 6, 9, 1]) returns *[4, 5]*. The procedure Check2 checks for errors using the

two equations mentioned above. Check2([1, 2, 3, 4, 5, 3, 2, 6, 9, 1, 4, 5]) returns *correct,* while Check2([1, 2, 5, 4, 3, 3, 2, 6, 9, 1, 4, 5]) returns *there is an error*, detecting the jump transposition error 3, 4, 5 → 5, 4, 3.

RSA ENCRYPTION

With the increasing popularity of the Internet, data security is becoming more and more important. Ron Rivest, Adi Shamir, and Len Adleman developed one of the most effective methods of encryption in the 1970s. Their method, which shall now be highlighted, is named RSA encryption in their honor. Gallian presents a description of the method as an application to external direct products in [Ga1]. As Gallian states, "the idea is based on the facts that there exist efficient methods for finding very large prime numbers (say about 100 digits long) and for multiplying large numbers, but no one knows an efficient algorithm for factoring large integers (say about 200 digits long)" [Ga1 p. 158-159].

A person wanting to receive encrypted information can select a pair of large prime numbers p and q and a number r such that $1< r < m = \text{lcm}(p-1, q-1)$ and $\gcd(r, m)=1$. Then that person announces that any numerical message M sent to him should be coded as $R = M^r \bmod n$, with $n = pq$. If n is large enough, it cannot be factored easily into pq. To decode the message the receiver must find an s such that $rs = 1 \bmod m$. Then the message can be decoded as $R^s \bmod n$.

Gallian shows why this method works using external direct products. He argues that an element of the form x^m in U(n) corresponds to one of the form (mx_1, mx_2) in $Z_{p-1} \oplus Z_{q-1}$ since U(n) is isomorphic to $U(p) \oplus U(q)$ which is isomorphic to $Z_{p-1} \oplus Z_{q-1}$. Thus $x^m \approx (mx_1, mx_2) = (u(p-1)x_1, v(q-1)x_2)$ for some u and v since $m = \text{lcm}(p-1, q-1)$. It follows that $x^m \approx (0, 0)$ in $Z_{p-1} \oplus Z_{q-1}$. Therefore $x^m = 1$ for all x in U(n). Thus $rs = 1 + tm$ for some t implies that $R^s = (M^r)^s = M^{rs} = M^{1 + tm} = M(M^m)^t = M$ in U(n). Hence the method is correct.

The author has implemented a simplified version of the RSA method in Maple. The procedure nrs takes two prime numbers and returns [n, r, s] where n, r, and s are as described above. In practice, the two prime numbers used should be about 100 digits long. This simple example works with smaller prime numbers selected by the user, and thus the code could be broken by factoring n and finding s. However, this example conveys the method used with larger numbers.

The procedure codemess is used to code a string of letters by converting letters A-Z and blanks represented by the underscore symbol _ into integers one through twenty-seven and hitting each integer with the procedure code. The procedure code takes a digit M and returns $M^r \bmod n$. A coded string of integers can be decoded using decodemess which hits each integer with the procedure decode and converts integers back to letters. The procedure decode takes a digit R and returns $R^s \bmod n$.

For example, consider the message "groups are powerful." The command NN:= nrs(nextprime(1000), nextprime(1200)) returns *NN:=[1211809, 1213, 17077]*. Then the message can be encoded with the command R:=codemess([g, r, o, u, p, s, _, a, r, e, _, p, o, w, e, r, f, u, l], NN[1], NN[2]),

which returns the list of integers R := [889934, 540600, 203332, 1101949, 1071174, 494240, 494871, 1, 540600, 222905, 494871, 1071174, 203332, 894511, 222905, 540600, 527606, 1101949, 304214]. R can be decoded with the command decodemess(R, NN[1], NN[3]), which returns *[g, r, o, u, p, s, _, a, r, e, _, p, o, w, e, r, f, u, l]*.

PERMUTATION PUZZLES

A permutation is a rule by which one can rearrange objects. Since many puzzles involve the rearrangement of different parts, they can be modeled and solved with permutation groups. Perhaps the most famous example is the Rubik's cube. It is not hard to see that rotating the six sides can generate every move of a Rubik's cube. By numbering the blocks as in figure 5, one can represent the group of moves for a Rubik's cube as a subgroup of S_{48}.

Using this numbering, the moves of the cube can be represented as P:= permgroup(48, {T, L, F, RT, R, B}) where T:=[[1, 3, 8, 6], [2, 5, 7, 4], [9, 33, 25, 17], [10, 34, 26, 18], [11, 35, 27, 19]], L:= [[9, 11, 16, 14], [10, 13, 15, 12], [1, 17, 41, 40], [4, 20, 44, 37], [6, 22, 46, 35]], F:= [[17, 19, 24, 22], [18, 21, 23, 20], [6, 25, 43, 16], [7, 28, 42, 13], [8, 30, 41, 11]], RT:= [[25, 27, 32, 30], [26, 29, 31, 28], [3, 38, 43, 19], [5, 36, 45, 21], [8, 33, 48, 24]], R:= [[33, 35, 40, 38], [34, 37, 39, 36], [3, 9, 46, 32], [2, 12, 47, 29], [1, 14, 48, 27]], and B:= [[41, 43, 48, 46], [42, 45, 47, 44], [14, 22, 30, 38], [15, 23, 31, 39],

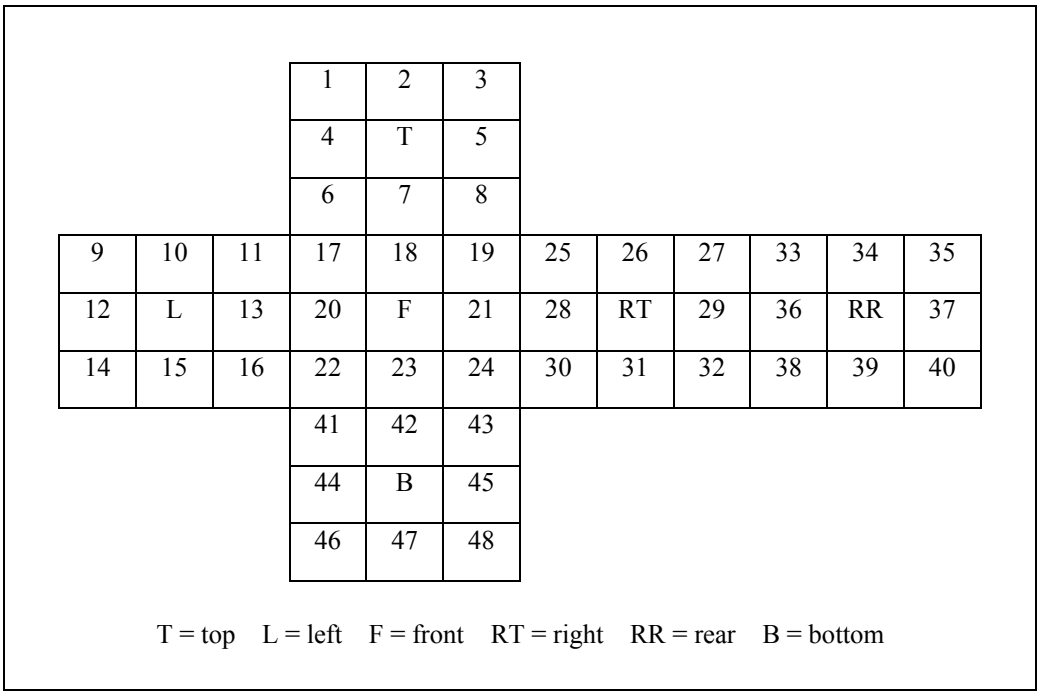

Figure 5. The numbered faces of the Rubik's cube.

[16, 24, 32, 40]]. Then grouporder(P) returns *43,252,003,274,489,856,000*, revealing the enormous number of possible states for the Rubik's cube.

Before discussing the Rubik's cube any further, a simpler example from [Ga1 p.103] will be considered. Suppose that one has the sliding puzzle in figure 6. The pieces are allowed to slide along the connecting lines. One can rotate the upper region formed by 1, 2, and 3, or the lower region formed by 1, 3, 4, 5, and 6. These two moves generate all possible moves of the puzzle. Hence the puzzle can be represented in Maple as the permutation group P:=permgroup(6, {[[1, 3, 4, 5, 6]], [[1, 2, 3]]}). Suppose that one wishes to put the puzzle into the form [[2, 3, 4]], i.e., perform this permutation on the original vertices. Obviously the only way to attempt this is to use some combination of the two mentioned rotations and their inverses. Thus to find a solution, one needs to represent [[2, 3, 4]] as a product of powers of [[1, 3, 4, 5, 6]] and [[1, 2, 3]]. The software package GAP [GAP1], which was designed specifically to perform computational group theory, has a built in function to do this, but Maple does not.

In chapter 2, Dimino's Algorithm is used to construct permutation groups by multiplying together generators. A simple adaptation of this can be used to factor permutations in small groups such as ⟨{[[1, 3, 4, 5, 6]], [[1,2,3]]}⟩. The algorithm factorperm, in FGv1.mpl runs through Dimino's algorithm until it finds the desired element. When the appropriate element is found, the algorithm returns the two elements that the algorithm multiplied together to form that element. One of these elements is a generator, but the other may be an element previously formed by multiplying together generators. The algorithm Factorp in FGv1.mpl uses factorperm recursively until a permutation is expressed as a product of generators. On the author's personal computer, [[2, 3, 4]] can be factored in less than one tenth of a second. The solution is [[2, 3, 4]] = $[[1, 3, 4, 5, 6]]^4[[1, 3, 2]][[1, 3, 4, 5, 6]]$. The algorithm is inefficient however for larger groups such as Sn for n>8. Of course, it depends on how quickly the element appears in the enumeration of the group

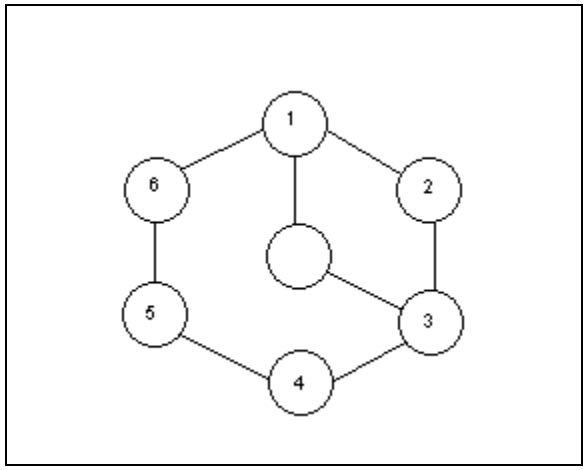

Figure 6. Sliding puzzle

elements. Unless the permutation appears as a product of very few generators early in the enumeration process, this algorithm has no chance with the Rubik's cube.

One would hope that the stabilizer chain as described in chapter 3 could be taken advantage of to factor a permutation. It is easy to use a stabilizer chain to represent a permutation as a product $s_1s_2...s_k$ where s_i is a coset representative of $G^{(i)}$ in $G^{(i-1)}$. This only requires a slight modification to the membership test algorithm. Recall that Maple stores coset representatives along with their inverses. Hence instead of checking for coset representatives s_i such that $s_i...s_1p$ fixes the i^{th} base point, one can check for inverses c_i such that $c_i...c_1p$ fixes the i^{th} base point. Then if p is in G, the algorithm will terminate with $c_k...c_1 = 1$. Thus $s_1...s_k = p$. Hence, one can store each s_i to represent a permutation as a product of coset representatives. This however does not solve the original problem.

The coset representatives for $G^{(i)}$ in $G^{(i-1)}$ are formed by taking products of the generators of $G^{(i-1)}$. Thus one could keep a record of the elements used to form each coset representative. Then each coset representative could be factored, allowing one to represent any element as a product of strong generators, i.e., the complete set of representatives for all of the stabilizers. This still does not solve the original problem since the strong generators may not be the original generators. The strong generators are formed using Schreier's lemma. Thus they are products of previous strong generators and coset representatives. This gives a way to factor a permutation into a product of the original generators. However, it is costly and inefficient.

The author has modified Maple's procedures `group/stabchain,` `group/addperm,` and `group/findorb` to keep two additional entries in the stabilizer chain. The fifth element of the stabilizer chains formed is a list of the factors of each strong generator. The sixth element is a list of the factors of each coset representative. The factors are stored as abstract letters representing the original generators. Another procedure then uses the sixth entry along with the coset representatives to factor a permutation. This method leads to the creation of extremely long words in the generators of the group. The permutation (2, 3, 4) is returned as a word in (1, 3, 4, 5, 6) and (1, 2, 3) and their inverses of length 104. Obviously this is not the shortest solution. Such methods have been shown to create words of length over 50,000 for the Rubik's cube [EP]. The version implemented in Maple cannot even construct the extremely long stabilizer chains for large permutation groups in a reasonable amount of time. Due to its inefficiency and the fact that it is adapted from copyrighted Maple procedures, the author has not published these procedures.

In 1994, Philip Osterlund released a set of functions for GAP to factor permutations. The file AbStab.g [Os1] was written for GAP version 3 and is incompatible with the current version 4. Osterlund's method reportedly formed solutions to the Rubik's cube with an average length of 1500 moves [EP]. Osterlund also includes a function shrink that can greatly reduce the size of the words formed. In an example from Osterlund's web page [Os2], his procedure is shown to factor a permutation into a word of length 1219, which is reduced to length 135 with shrink. This is an enormous improvement over 50,000 moves.

The author has attempted to code a version of AbStab.g in Maple. Maple was not designed to handle groups as easily as GAP. The author has written several procedures to perform some of the built in functions that GAP contains. On the author's PC, the routine MakeAbStabChain can form the abstract stabilizer chain for the Rubik's cube group in less than twenty seconds. The routine Factorperm in AbStab.mpl can factor a random permutation in a few seconds. The author has produced words of length less than 1000 for every example that he has tried. However, there are 43,252,003,274,489,856,000 possible permutations.

Instead of implementing Osterlund's shrink, the file AbStab.mpl contains a procedure tremble similar to one described in [EP] to reduce words. This procedure implements a very simple concept. The lengths of the words formed by Factorperm vary greatly from permutation to permutation. Thus the more words that one factors, the greater the chance is to produce a shorter word. Thus the procedure tremble takes a list of short words, i.e., words of length one or two in the generators, and factors the permutation obtained from sp where s is a short word and p is the desired permutation. The result can be multiplied by s^{-1} to obtain a word for p. The more short words used, the greater the chance of forming a shorter word for p. The procedure has reduced words to lengths between 100 and 220 for examples tried by the author. Using approximately 200 short words takes about five minutes on the author's computer to reduce a word.

The procedures in AbStab.g or AbStab.mpl work using basically the same ideas as Maple's `group/stabchain` with some necessary modifications. Maple's `group/stabchain` is not designed to enable one to factor a permutation. Thus, no abstract generators are stored, and generators are added to the chains one at a time. To factor a permutation efficiently, one needs to assign abstract generators to represent each of the original generators. Then the strong generators and coset representatives or their inverses need to be represented as words in the abstract generators. To ensure that these words do not grow exponentially, it is necessary to form all of the Schreier generators for each stabilizer before moving to the next stabilizer. Then the generators can be sorted according to their abstract generator word lengths, and redundant generators with long words can be discarded. Hence, shorter words will be used to construct the rest of the stabilizers.

The main procedure to form a stabilizer chain in AbStab.mpl is MakeAbStabChain. The input for this procedure is the degree of the permutation group, the generators, abstract generators, and two optional arguments. The first optional argument is a base. Thus if one knows a short base for the group, it can be used. If anything is entered for the last optional argument, then the procedure assumes that the generators are disjoint cycles. If only four arguments are used, then the procedure assumes that the generators are permutation lists. The procedure uses MakeStab and FirstStabilizer to form the first stabilizer for the group entered. Then the procedure is called recursively to complete the stabilizer chain. The output is the generators, abstract generators, the orbit of the first base point under the group, a table trans that will be described soon, and the stabilizer chain for the next stabilizer. Thus, the procedure builds a nested list similar to `group/stabchain,` but the information is more suitable to factor permutations.

The procedure MakeStab forms the orbit a base point b_i under $G^{(i-1)}$, and the table trans. The table trans stores the information necessary to trace from an orbit point to the base point, giving a permutation p and its abstract word $\bar{p}$ such that $p[x] = b_i$ for a point x in the orbit. Thus p is the inverse of a coset representative, which is necessary to form the Schreier generators. If a base has not been defined by the main procedure then MakeStab defines it as the set of points moved by $G^{(i-1)}$. This set is calculated by the author's procedure movedpoints. The procedure forms the orbit by hitting each point already in the orbit with each generator and its inverse. Let the generators of $G^{(i-1)}$ be the set gens. If $(gens[m_1])[k_1] = j_1$ for some k_1 in the orbit and j_1 is not in the orbit, then j_1 is added to the orbit, and $trans[j_1]$ is set to $-m_1$, implying that $(gens[m_1])^{-1}[j_1] = k_1$. If $(gens[m_2])^{-1}[k_2] = j_2$ for some k_2 in the orbit and j_2 is not in the orbit, then j_2 is added to the orbit, and $trans[j_2]$ is set to m_2, implying that $(gens[m_2])[j_2] = k_2$. Again, these will be used to trace an orbit point back to b_i.

FirstStabilizer finds the generators for a stabilizer. The procedure uses Gentable to create a complete set of Schreier generators for the stabilizer. Gentable calls a procedure LeftSchreier to form the inverses of the coset representatives by tracing trans as described above. These elements are returned as permutations and abstract words in the original generators. This creates an extremely redundant list of generators. The redundancies are eliminated with MinGenSet. The idea is to form the stabilizer subgroup by adding permutations one at a time, not adding any permutations that are already in the group. This is exactly what `group/stabchain` does, but it doesn't return any abstract generators. Thus, the author has made a few simple modifications to `group/stabchain` and `group/addperm` to keep a list of abstract generators. These are found in AbStab.mpl as the procedures stabchain and addperm.

To ensure that the generators with the shortest abstract generators are used, it is necessary to perform a parallel sort of the generators and abstract generators with the author's procedure psort. This procedure sorts the abstract generators according to length, putting the shortest in the front. It also moves the generators accordingly so that the i^{th} generator corresponds to the i^{th} abstract generators. This forces the stabilizer subgroup to be formed from the shortest abstract generators. This is the key idea of Osterlund's that reduces the length of words, preventing exponential growth. Before returning the final set of generators, FirstStabilizer performs another parallel sort to ensure that the shorter words will be used first in future calculations.

These procedures work together recursively to form a complete stabilizer chain with all of the information necessary to factor a permutation. The parallel sorts ensure that the words do not grow exponentially in length. This leads to quicker formations of stabilizer chains and shorter factors. It reduces solutions to the cube by about 40,000 moves as compared to the procedure without parallel sorts.

The file AbStab.mpl also contains a procedure RCfindperm that can be used to find the permutation necessary to solve an actual Rubik's cube. Once the user has defined the colors of the cube, the

procedure takes the input of all colors for blocks 1 to 48 and determines the permutation necessary to solve the cube. Then Factorperm can be used to find a solution to the cube.

The author has used these procedures to physically solve a Rubik's cube. The colors and sides on this particular cube were entered as FT:=b, TP:=r, LF:=y, RT:=w, BK:=g, and BT:=o. Then L:= [o, g, b, r, r, y, r, g, b, g, r, b, o, y, r, y, g, b, r, w, g, r, b, o, w, w, w, w, y, w, o, o, r, y, w, o, w, y, y, g, b, o, g, y, g, o, b, b] set the colors that were in slots one to forty-eight. Then q1:= RCpermfind(L) returned the permutation that solves this cube, *q1 := [43, 37, 19, 2, 5, 9, 7, 33, 24, 34, 1, 21, 45, 14, 4, 11, 35, 18, 3, 31, 36, 6, 23, 48, 27, 26, 25, 29, 15, 32, 47, 41, 8, 12, 30, 44, 28, 16, 13, 40, 17, 42, 38, 10, 39, 46, 20, 22]*.

After finding the necessary permutation q1, the stabilizer chain for the group was defined as pg:= MakeAbStabChain(48, Gens, [[Left], [Front], [Right], [Rear], [Bot], [Top]], [1, 6, 3, 8, 21, 23, 26, 5, 29, 19, 7, 24, 25, 28, 31, 18, 4, 2], 1). Gens is defined by AbStab.mpl as the before mentioned generators corresponding to the rotations of the six sides of the cube. On the author's PC, pg was constructed in 14.9 seconds. After constructing pg, x:=Factorperm(48,p,pg):nops(x);evalb(checker(x,pg[1],pg[2],48)=p); returned *669* and *true* in less than one second. This means that x has been stored as a list of length 669, and checker(x, pg[1], pg[2], 48) = p confirms that the factorization is correct. Then a list of short words was created by typing R:=addinvs(pg[2]) and SW:=ShortWords(R, pg, 48, 1, 12, 1, 12). The list SW created by this command contains 127 abstract elements of length one or two in the generators and their inverses. In 32 seconds, the command d:=Tremble(SW, pg, 48, x, q1): nops(d); evalb(checker(d, pg[1], pg[2], 48)=q1) returned *111* and *true*. This gives the correct solution d:= [1/Rear, 1/Left, Bot, Front, 1/Bot, 1/Front, 1/Bot, 1/Right, Bot, Right, 1/Bot, 1/Bot, 1/Top, 1/Rear, Top, 1/Bot, 1/Top, Rear, Top, Front, Bot, 1/Front, 1/Top, 1/Rear, Top, 1/Bot, 1/Top, Rear, Top, Front, Bot, 1/Front, Bot, Front, Bot, 1/Front, 1/Bot, 1/Right, 1/Bot, Right, 1/Top, 1/Rear, Top, 1/Bot, 1/Top, Rear, Top, Front, Bot, 1/Front, 1/Bot, 1/Right, 1/Top, Rear, Top, 1/Bot, 1/Top, 1/Rear, Top, Bot, Right, 1/Bot, 1/Bot, Front, 1/Bot, Front, 1/Right, 1/Bot, Right, 1/Front, 1/Bot, 1/Bot, 1/Bot, Front, Front, 1/Right, 1/Bot, Right, 1/Front, 1/Front, 1/Top, 1/Rear, Top, Right, 1/Top, 1/Rear, Top, Right, 1/Top, 1/Rear, Top, Right, Right, Rear, 1/Top, 1/Rear, Right, Rear, 1/Top, 1/Rear, 1/Top, 1/Rear, Top, Right, 1/Left, 1/Top, Left, 1/Front, 1/Front, 1/Left, 1/Rear]. The author correctly implemented this sequence of moves to solve the cube in approximately fifteen minutes.

The procedures in AbStab.mpl provide an extremely short solution almost instantly to the sliding puzzle. The commands pg:= MakeAbStabChain(6, [[[1, 3, 4, 5, 6]], [[1, 2, 3]]], [[a1], [a2]], [], 1): x:=Factorperm(6, [1, 3, 4, 2, 5, 6], pg); return the solution x:= [1/*a*1, 1/*a*2, *a*1] in less than one tenth of a second. Clearly, this is the shortest possible solution for this problem. Hence the procedures in AbStab.mpl can be used to efficiently solve a wide range of permutation puzzles in a short amount of time.

The most effective method known for solving permutation puzzles is described by Egner and Püschel in [EP]. They have a rather complicated method for constructing the shortest base possible and reducing words as they build the stabilizer chains. They claim that their algorithm can produce solutions to the Rubik's cube with average length 60. Egner himself stated in a GAP forum posting that "The program itself is not really fit for publication and it has been implemented for the older version GAP3 and is not

available for the new version GAP4" [Eg]. Thus it is possible to solve large permutation puzzles with short solutions using carefully constructed stabilizer chains, but it is a difficult process. The solutions are not a great deal shorter than those obtained by AbStab.mpl or AbStab.g.

CONCLUSION

This thesis has shown ways to deal with small finite groups, permutation groups, and finitely presented groups. The final chapter has shown three simple examples of ways to use computers and group theory to solve real world problems. As previously stated, this is not a comprehensive survey of computational group theory. It is merely an introduction into some of the amazing capabilities of this branch of mathematics. Mathematicians have made incredible advances in this field since the first computers became available. Computers have already enabled mathematicians to solve some problems that would have otherwise been impossible. The sheer computational power of today's processors allow for an incredible amount for computations to be performed in a small amount of time. Computers will never replace the incredible adaptive abilities of the human mind, but one would be foolish not to incorporate these powerful tools into his problem solving.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Bu] Butler, G., *Fundamental Algorithms for Permutation Groups*, Springer-Verlag, Berlin, 1991.

[BC] Butler, G. and J. Cannon, Computing in Permutation and Matrix Groups III: Sylow Subgroups, *Journal of Symbolic Computation,* Volume 8 (1989): 241-252.

[Ca] Cameron, Peter, *Permutation Groups,* Cambridge University Press, Cambridge, 1999.

[Cai] Cairns, Stewart Scott, *Introductory Topology,* The Ronald Press Company, New York, 1961.

[CD] Cannon, J., L. Dimino, et. al. Implementation and Analysis of the Todd-Coxeter Algorithm, *Mathematics of Computations,* volume 27 (July 1973): 463-490.

[CG] Char, B., O. Geddes, et al. *Maple V Library Reference Manual*, Springer-Verlag, NewYork, 1991.

[DL] Dubinsky, E. and U. Leron, *Learning Abstract Algebra with ISETL,* Springer-Verlag, 1994.

[Eg] Egner, S., Re: GAP commands equivalent to magma's InverseWordMap and so. Posting to GAP forum (9-19-2000) http://turnbull.mcs.stand.ac.uk/~gap/Info/forum.html, 9-30-2001.

[EP] Egner, S., and Markus Püschel, Solving Puzzles Related to Permutation Groups, *Proceedings ISSAC,* (1998): 186-193.

[Ga1] Gallian, Joseph A., *Contemporary Abstract Algebra* (4^{th} ed.), Houghton Mifflin Company, Boston, 1998.

[Ga2] Gallian, Joseph A., Error Detection Methods, *ACM Computing Surveys,* Volume 28 (1996): 504-517.

[GAP1] *GAP 4.2 Manual*, Internet, http://www-gap.dcs.st-andrews.ac.uk/~gap/Info4/manual.html, 10-4-2001.

[GAP2] *Subgroup Presentations,* Internet, http://www-gap.dcs.st-andrews.ac.uk/~gap/Manual4/ref/C045S003.htm#I13, 11-14-2001.

[Hu] Hungerford, Thomas, *Algebra*, Springer-Verlag, New York, 1974.

[Le1] Leech, John, Coset Enumeration, *Computational Problems in Abstract Algebra,* Ed. John Leech, Pergamon Press, 1970. 21-35.

[Le2] Leech, John, Computer Proof of Relations in Groups, *Topics in Group Theory and Computation*, Ed. Michael P. J. Curran, Academic Press, New York, 1977. 38-61.

[Mc] McLain, D.H., An Algorithm for Determining Defining Relations of a Subgroup, *Glasgow Math. Journal*, Volume 18 (1977): 51-56.

[Me] Mendelsohn, N.S., Defining Relations for Subgroups of Finite Index of Groups with a Finite Presentation, *Computational Problems in Abstract Algebra*, Ed. John Leech, Pergamon Press, New York, 1970. 43-44.

[Ne] Neubuser, J., An Elementary Introduction to Coset Table Methods in Computational Group Theory, *Group St. Andrews 1981*, Ed. C.M. Campbell and E.F. Robertson, Cambridge University Press, Cambridge, 1982. 1-45

[Os1] Osterlund, Philip, AbStab.g, (1994): Internet, http://www.math.umn.edu/~osterlu/AbStab/AbStab.g. 10-4-2001.

[Os2] Osterlund, Philip, AbStab.dem, (1994): Internet, http://www.math.umn.edu/~osterlu/AbStab/AbStab.dem. 10-4-2001.

[Ro] Rose, J.S., *A Course on Group Theory*, Cambridge University Press, Cambridge, 1978.

[TCx] Todd, J. A., and H. S. M. Coxeter, A Practical Method for Enumerating Cosets of a Finite Abstract Group, *Proceedings of the Edinburgh Mathematical Society*, Volume 5 (1936): 26-34.

[Tr] Trotter, H.F., A Machine Program for Coset Enumeration, *Canadian Math. Bulletin,* Volume 7 (July 1964): 357-368.

[WaM] Waterloo Maple Inc., *Maple V Release 5.1*, 1998.

[Wi] Winters, S., Error-Detecting Schemes Using Dihedral Groups, *UMAP Journal*, Volume 11 (1990): 299-308.

APPENDIX

APPENDIX

MAPLE ALGORITHMS

FGv1.mpl

```
print(` Finite Group Package loaded: `);
printf(` Closed, Assoc, Identity, Inverse, Inverses, Isgp, Comm, Isidentity\n`);
printf(` Zmodn, Umodn, sn, Dsum, q8, pgp, t12, Grelgroup, Dimino\n`);
printf(` Ctable, newf, elpow, elord ,elords, sbgp, subgps, subgpsord\n`);
printf(` cyca, iscyclic, Center, Centralizer, Normalizer, Conjugate, Lcoset, Rcoset\n`);
printf(` HK, Lcosets, GmodH, isnorm\n`);
printf(` Homom, kernel, image, invimage, isom, Factorperm\n`);
printf(` procedures to find finite abelian groups loaded\n`);
printf(` cpint, cp, partx, fntabel, numbfntabel\n`);

with(group):with(combinat):

Assoc:=proc(G,f,m)
# tests associativity
 local i,j,k,x,y,z,ord:
 ord:=nops(G);
 for i from 1 to ord do
  for j from 1 to ord do
   for k from 1 to ord do
    x:=f(G[i],G[j]):
    y:=f(x,G[k]):
    z:=f(G[j],G[k]):
    x:=f(G[i],z):
    if(x<>y) then
     if(nargs=2) then RETURN(`not associative`); fi;
     printf(`this set with this operation is not associative\n`);
     printf(`since ( %a * %a)* %a = %a\n`,G[i],G[j],G[k],y);
     printf(`and  %a * ( %a * %a )= %a\n`,G[i],G[j],G[k],x);
     RETURN();
    fi;
   od:
  od:
 od:
 if(nargs=2) then RETURN(`associative`); fi;
 printf(`this set with this operation is asscociative\n`);
end:

Center:=proc(G,f)
# finds the center of a group
 local i,j,x,C;
 C:=[];
 for i from 1 to nops(G) do
  x:=0:
  for j from 1 to nops(G) do
   if(f(G[i],G[j])<>f(G[j],G[i])) then break; fi;
   x:=j:
  od;
  if(x=nops(G)) then C:=[op(C),G[i]]; fi;
```

```
 od;
 RETURN(C);
end:

Centralizer:=proc(H,G,f)
#finds the centralizer of H in G
 local i,j,x,C,n;
 n:=nops(H);
 C:=[];
 for i from 1 to nops(G) do
  x:=0:
  for j from 1 to n do
   if(f(G[i],H[j])<>f(H[j],G[i])) then break; fi;
   x:=j:
  od;
  if(x=n) then C:=[op(C),G[i]]; fi;
 od;
 RETURN(C);
end:

Closed:=proc(G,f,x)
# determines whether or not a set (or list), G, with a given operation, f,
# is closed
 local i,j,m,n,y,z;
 y:=nops(G);
 for i from 1 to y do
  for j from 1 to y do
   z:=member(f(G[i],G[j]),G):
   if (z=false) then
    if(nargs=3) then
     printf(` this set with this operation is not closed since\n`);
     printf(`%a( %a,%a)=%a\n`,f,G[i],G[j],f(G[i],G[j]));
     RETURN();
    fi;
    RETURN(`not closed`);
   fi;
  od;
 od;
 if(nargs=2) then RETURN(`closed`); fi;
 RETURN(`This set with this operation is closed`);
end:

Comm:=proc(G,f,x)
# tests commutativity
 local i,j,m,n,ord:
 ord:=nops(G):
 for i from 1 to ord do
  for j from 1 to ord do
   if(f(G[i],G[j])<>f(G[j],G[i])) then
    if(nargs=2) then RETURN(`not commutative`); fi;
    printf(` this set with this operation is not commutative\n`);
    printf(` since %a * %a = %a and %a * %a = %a\n`,
    G[i],G[j],f(G[i],G[j]),G[j],G[i],f(G[j],G[i]));RETURN();
   fi;
  od;
```

```
 od;
  if(nargs=2) then RETURN(`commutative`); fi;
  printf(` this set with this operation is commutative\n`);
end:

Conjugate:=proc(x,H,G,f)
#find xHx^-1
 local ainv,i,y,z,m,Conj;
 m:=member(x,G):
 if(m=false) then RETURN(` ERROR: not in group`);fi;
 Conj:={};
 ainv:=Inverse(x,G,f):
 for i from 1 to nops(H) do
  y:=f(f(x,H[i]),ainv):
  z:=member(y,Conj):
  if (z=false) then
   Conj:={op(Conj),y}:
  fi;
 od;
 Conj:=convert(Conj,list);
 RETURN(Conj);
end:

cp:=proc(LL)
 local i,j,A,x,n,m;
 n:=1;
 m:=nops([indices(LL)]):
# sets up an array of orders to pass to cpint
# also calculates n, the number of tuples in A
 x:=array(1..m):
 for i from 1 to m do
  x[i]:=nops([indices(LL[i])]):
  n:=n*nops([indices(LL[i])]):
 od:
 A:=cpint(x):
# use A to find desired result
 for i from 1 to n do
  for j from 1 to m do
   A[i,j]:=LL[j][A[i,j]]:
  od;
 od;
 RETURN(A);
end:

cpint:=proc(L)
#takes an array of integers and return an array where each row
#is a nops(L) tuple with entries 1..L[i]
# i.e. it does Cartesian product on sets of integers
 local i,j,A;
 global n;
 n:=1;
# determines the row size, n,  of the array
 for i from 1 to nops([indices(L)]) do
  n:=n*L[i]:
 od:
```

```
# set up array and initialize row 1
 A:=array(1..n,1..nops([indices(L)])):
 for i from 1 to nops([indices(L)]) do
  A[1,i]:=1;
 od;
# perform Cartesian product
 for i from 2 to n do
  A[i,1]:=(A[i-1,1]+1) mod (L[1]+1):
  for j from 2 to nops([indices(L)]) do
   A[i,j]:=A[i-1,j]:
  od;
  for j from 1 to nops([indices(L)])-1 do
   if(A[i,j]=0) then
    A[i,j+1]:=(A[i-1,j+1]+1) mod (L[j+1]+1):
    A[i,j]:=1;
   fi;
  od;
 od;
RETURN(A);
end:

Ctable:=proc(G,f)
# construct a Cayley Table for G and f
 local i,j,n:
 global CT:
 n:=nops(G):
 CT:=array(1..n,1..n):
 for i from 1 to n do
  for j from 1 to n do
   CT[i,j]:=f(G[i],G[j]):
  od:
 od:
print(` CT is loaded`);
end:

cyca:=proc(a,G,f)
# finds the cyclic subgroup <a>
 local i,y,z,id,X:
 id:=Identity(G,f);
 z:=member(a,G):
 if(z=false) then RETURN(` Error that element is not in the set`); fi;
 y:=a:
 X:=[]:
 for i from 1 to nops(G) while(y<>id) do
  y:=elpow(a,i,G,f):
  X:=[op(X),y];
 od;
 RETURN(X);
end:

Dimino:=proc(L)
 local s,elt,rep_pos,order,G,i,j,g,porder;
 order:=1;
 G:=[[]];
 g:=L[1];
```

```
 while (g<>[]) do
  order:=order+1;
  G:=[op(G),g];
  g:=mulperms(g,L[1]);
 od;
 for i from 2 to nops(L) do
  if(member(L[i],G)=false) then
   porder:=order;
   order:=order+1;
   G:=[op(G),L[i]];
   for j from 2 to porder do
    order:=order+1;
    G:=[op(G),mulperms(G[j],L[i])];
   od;
   rep_pos:=porder+1;
   while(rep_pos<=order) do
    for s in L do
     elt:=mulperms(G[rep_pos],s);
     if(member(elt,G)=false) then
      order:=order+1;
      G:=[op(G),elt];
      for j from 2 to porder do
       order:=order+1;
       G:=[op(G),mulperms(G[j],elt)];
      od;
     fi;
    od;
    rep_pos:=rep_pos+porder;
   od;
  fi;
 od;
 RETURN(G);
end:

dop:=proc(x,y,L2)
 local i,j,k;
 k:=[];
 for i to nops(L2) do
  k:=[op(k),L2[i](x[i],y[i])];
 od;
 RETURN(k);
end:

Dsum:=proc(L,L2)
local T1,t2,i,j,k,ord,l,L1;
 global DS, fds;
 L1:=array(1..nops(L));
 for i to nops(L) do
  L1[i]:=LtoA(L[i]);
 od;
 T1:=cp(L1);
 ord:=1; l:=nops(L);
 for i to l do ord:=ord*nops(L[i]) od;
 DS:=array(1..ord);
 for i to ord do
```

```
 t2:=[];
 for j to l do
  t2:=[op(t2),T1[i,j]];
 od;
 DS[i]:=t2;
od;
DS:=['DS[i]'$'i'=1..ord];
fds:=(x,y)->dop(x,y,L2);
print(`DS and fds loaded`);
end:

elord:=proc(a,G,f)
# finds the order of an element a in G
 local i,x,w,id:
 i:=member(a,G):
 id:=Identity(G,f);
 if(i=false) then RETURN(`Error that element is not in the set`); fi;
 w:=1:
 x:=a:
 for i from 1 to nops(G) while(x<>id) do
  x:=f(x,a):
  w:=w+1:
 od:
 RETURN(w);
end:

elords:=proc(G,f)
# stores the orders of each element in G in a list ORDS
# ORDS[i]:=the order of G[i]
 local i,n,ORDS,w:
 n:=nops(G):
 ORDS:=[];
 for i from 1 to n do
  w:=elord(G[i],G,f):
  ORDS:=[op(ORDS),w];
 od:
 RETURN(ORDS);
end:

elpow:=proc(a,n::integer,G,f)
# find a^n for any integer n and any a in G
 local i,x,ans,ainv:
 ans:=Identity(G,f);
 x:=member(a,G):
 if(x=false) then RETURN(` Error that element is not in the set`); fi;
  if(n>0) then
   for i from 1 to n do
    ans:=f(ans,a):
   od:
  fi:
 if(n<0) then
  ainv:=Inverse(a,G,f):
  for i from 1 to -n do
   ans:=f(ans,ainv):
  od:
```

```
 fi:
 RETURN(ans);
end:

Factorp:=proc(L,t)
 local d,fp; fp:=[];
 d:=factorperm(L,t);
 while d[1]<>[] do
  fp:=[d[2],op(fp)];
  d:=factorperm(L,d[1]);
 od;fp:=[d[2],op(fp)];
 RETURN(fp);
end:

factorperm:=proc(L,t)
 local s,elt,rep_pos,order,G,i,j,g,h,porder;
  if(L[1]=t) then RETURN([],L[1]); fi;
  order:=1; G:=[[]];
  g:=L[1];
  while (g<>[]) do
   order:=order+1;
   G:=[op(G),g];
   h:=(mulperms(g,L[1]));
   if(h=t) then RETURN(g,L[1]); fi;
   g:=h;
  od;
  for i from 2 to nops(L) do
   if(member(L[i],G)=false) then
    porder:=order;
    order:=order+1;
    if(L[i]=t) then RETURN([],L[i]); fi;
    G:=[op(G),L[i]];
    for j from 2 to porder do
     order:=order+1;
     g:=mulperms(G[j],L[i]);
     if(g=t) then RETURN(G[j],L[i]); fi;
     G:=[op(G),g];
    od;
    rep_pos:=porder+1;
    while(rep_pos<=order) do
    for s in L do
     elt:=mulperms(G[rep_pos],s);
     if(elt=t) then RETURN(G[rep_pos],s); fi;
     if(member(elt,G)=false) then
      order:=order+1;
      G:=[op(G),elt];
      for j from 2 to porder do
       order:=order+1; g:=mulperms(G[j],elt);
       if(g=t) then RETURN(mulperms(G[j],G[rep_pos]),s); fi;
       G:=[op(G),g];
      od;
     fi;
    od;
    rep_pos:=rep_pos+porder;
   od;
```

```
  fi;
 od;
 RETURN([],`not in group`);
end:

fntabel:=proc(n)
#calculates an array y which has rows made up of
# the elementary divisors of a finite abelian group of order n
 local i,x,L,y;
 x:=op(2,numtheory[ifactors](n));
 L:=array(1..nops(x));
 for i from 1 to nops(x) do
  L[i]:=partx(x[i][2],x[i][1]):
 od:
 RETURN(cp(L));
end:

GmodH:=proc(H,G,f,g)
# forms the factor group G/H with op g(aH,bH)=(abH)
 local i,y,z,j,xH,X;
 global GMODH;
 X:={};
 for j from 1 to nops(G) do
  xH:={};
  for i from 1 to nops(H) do
   y:=f(G[j],H[i]):
    xH:={op(xH),y}:
  od;
   X:={op(X),xH};
 od;
  GMODH:=X;
  g:=(a,b)->Lcoset(f(a[1],b[1]),H,G,f);
print(GMODH);
end:

Grelgroup:=proc(gen,rel,f)
 local gg,subg;
 global G;
 gg:=grelgroup(gen,rel);
 subg:=subgrel({z=op(1,rel)},gg);
 G:={};
 G:=cosets(subg):
 f:=(a,b)->cosrep(`group/mulword`(a,b),subg)[2]:
 print(`G loaded`);
end:

HK:=proc(H,K,G,f)
#find HK
 local i,j,y,z,HoK;
 HoK:={};
 for i from 1 to nops(H) do
  for j from 1 to nops(K) do
   y:=f(H[i],K[j]):
    HoK:={op(HoK),y}:
  od;
```

```
 od;
 RETURN(HoK);
end:

Homom:=proc(f,G1,op1,G2,op2)
# determines if f is a homomorphism G1->G2
 local i,j;
 for i from 1 to nops(G1) do
  for j from 1 to nops(G1) do
   if(f(op1(G1[i],G1[j]))<>op2(f(G1[i]),f(G1[j]))) then
    printf(`%a(%a*%a)=%a but %a(%a)*%a(%a)=%a\n`,f,G1[i],G1[j],
   f(op1(G1[i],G1[j])),f,G1[i],f,G1[j],op2(f(G1[i]),f(G1[j])));
   RETURN(`Does not preserve operations`);
   fi;
  od;
 od;
 RETURN(`homomorphism`)
end:

Identity:=proc(G,f)
# finds the identity of a set G under the operation f or returns noid
 local ct,i,j,x,y,n,id:
 id:=noid:
 n:=nops(G):
 for i from 1 to n do
  ct:=0:
  for j from 1 to n do
   x:=f(G[i],G[j]):
   if(x<>G[j]) then break fi;
   x:=f(G[j],G[i]):
   if(x<>G[j]) then break fi;
   ct:=ct+1:
  od:
  if(ct=n) then id:=G[i]; break fi;
 od:
 RETURN(id);
end:

image:=proc(f,G1,op1)
# finds the image of f
 local i,x,y;
 global Imf;
 Imf:={};
 for i from 1 to nops(G1) do
  y:=f(G1[i]):
  x:=member(y,Imf):
  if(x=false) then Imf:={op(Imf),y} fi;
 od;
 Imf:=convert(Imf,list):
 RETURN(Imf);
end:

Inverse:=proc(a,G,f,m)
# searches for the inverse of an element a
 local id,i,x,y,n,z,ainv:
```

```
 z:=member(a,G):
 if(z=false) then
 RETURN(`Error: that element is not in that set`) fi;
 n:=nops(G):
 id:=Identity(G,f):
 ainv:=noinv:
 for i from 1 to n do
  x:=f(G[i],a):
  y:=f(a,G[i]):
  if(x=id and y=id) then ainv:=G[i]; break fi;
 od:
 if(ainv<>noinv) then
  if(nargs=3) then RETURN(ainv); fi;
  printf(` %a inverse is %a since %a * %a = %a * %a = %a\n`,a,ainv,a,ainv,ainv,a,x);
 fi;
 if(ainv=noinv) then
  if(nargs=3) then RETURN(`noinv`); fi;
  printf(` %a has no inverse in this set\n`,a);
 fi;
end:

Inverses:=proc(G,f,x)
# tests if every element in a set has an inverse
 local i,m,n,id,ainv:
 id:=Identity(G,f);
 n:=nops(G):
 if(id=noid) then RETURN(`no identity implies no inverses`); fi;
 for i from 1 to n do
  ainv:=inverse(G[i],G,f):
  if(ainv=noinv) then
   if(nargs=2) then RETURN(`no inverses`); fi;
   printf(` %a has no inverse\n`,G[i]);
   RETURN();
  fi;
 od;
 if(nargs=2) then RETURN(`inverses`); fi;
 RETURN(`Every element in this set has an inverse`);
end:

invimage:=proc(H,f,G1)
# finds all x in G1 such that f(x) is in H
 local i,x;
 global finvH;
 finvH:={}:
 for i from 1 to nops(G1) do
  x:=member(f(G1[i]),H):
  if(x=true) then finvH:={op(finvH),G1[i]} fi;
 od;
 finvH:=convert(finvH,list):
 RETURN(finvH);
end:

iscyclic:=proc(G,f)
# uses elords to determine if a group is cyclic
 local i,ans,x,ORDS;
```

```
 ORDS:=elords(G,f):
 x:=0:
 for i from 1 to nops(G) do
  if(ORDS[i]=nops(G)) then
   printf(` this group is cyclic generated by %a\n`,G[i]);
   x:=1:
   break;
  fi;
 od;
 if(x=0) then print(` not cyclic`); fi;
end:

Isgp:=proc(G,f,m)
# determines if a set with an operation is a group
 local k,x,y,z,n,ansC,ansA,id,ansI:
 ansC:=Closed(G,f):
 ansA:=Assoc(G,f):
 id:=Identity(G,f):
 ansI:=Inverses(G,f):
 y:=notmonoid:
 z:=notgrp:
 for k from 1 to 1 do
  if(ansC<>closed or ansA<>associative) then
   if(nargs=2) then RETURN(`not a group`); fi;
   x:=notsemigp;
   break;
  fi;
  x:=semigrp:
  if(id=noid) then
   if(nargs=2) then RETURN(`not a group`); fi;
   y:=notmonoid;
   break;
  fi;
  y:=monoid:
  if(ansI<>inverses) then
  if(nargs=2) then RETURN(`not a group`); fi;
  z:=notgrp; break fi;
  if(nargs=2) then RETURN(`Group`); fi;
  z:=grp:
 od:
 printf(` The operation is %a and %a.\n`,ansC,ansA);
 printf(` The set has %a and %a\n`,id,ansI);
 printf(` Hence this set with this operation is %a, %a, and %a\n`,x,y,z);
end:

Isidentity:=proc(a,G,f)
# tests if a is the f identity of G
 local z,ct,j,x,y,n:
  ct:=0:
  n:=nops(G):
  z:=member(a,G):
  if(z=false) then RETURN(` Error: that element is not in the set`) fi;
  for j from 1 to n do
   x:=f(a,G[j]):
   if(x<>G[j]) then
```

```
    printf(` %a is not the identity since %a * %a = %a\n`,a,a,G[j],x);
    break;
   fi;
   x:=f(G[j],a):
   if(x<>G[j]) then
    printf(` %a is not the identity since %a * %a = %a\n`,a,G[j],a,x);
    break;
   fi;
   ct:=ct+1:
  od:
  if(ct=n) then
   printf(` %a is the identity of this set\n`,a);
  fi;
end:

isnorm:=proc(H,G,f)
# checks to see if a set H is normal
 local i,Conj;
 for i from 1 to nops(G) do
  Conj:=convert(Conjugate(G[i],H,G,f),set):
  if(Conj<>convert(H,set)) then
  printf(` %a %a %a^-1 = %a\n`,G[i],H,G[i],Conj);
 RETURN(`is not normal`);
 fi;
 od;
 RETURN(`normal`);
end:

isom:=proc(f,G1,op1,G2,op2)
# determines whether f is isomorphism, epimorphism, monomorphism, or homomorphism
 local i,x,y,z,id;
 id:=Identity(G1,op1);
 x:=Homom(f,G1,op1,G2,op2):
 if(x<>homomorphism) then RETURN(x); fi;
 y:=kernel(f,G1,op1,G2,op2);
 if(y=[id]) then
  x:=monomorphism:
 fi;
 z:=image(f,G1,op1):
 if(nops(z)=nops(G2) and x=monomorphism) then RETURN(`isomorphism`); fi;
 if(nops(z)=nops(G2) and x<>monomorphism) then RETURN(`epimorphism`); fi;
 if(nops(z)<>nops(G2)) then RETURN(x); fi;
end:

kernel:=proc(f,G1,op1,G2,op2)
# finds all elements of G1 such that f(a) = Identity(G2,op2)
 local i,id;
 global KER;
 id:=Identity(G2,op2);
 KER:={}:
 for i from 1 to nops(G1) do
  if(f(G1[i])=id) then KER:={op(KER),G1[i]}; fi;
 od;
 KER:=convert(KER,list):
 RETURN(KER);
```

```
end:

Lcoset:=proc(x,H,G,f)
#find xH
 local i,y,z,m,xH;
 m:=member(x,G):
 if(m=false) then RETURN(` ERROR: not in group`);fi;
 xH:={};
 for i from 1 to nops(H) do
  y:=f(x,H[i]):
  z:=member(y,xH):
  if (z=false) then
   xH:={op(xH),y}:
  fi;
 od;
 RETURN(xH);
end:

Lcosets:=proc(H,G,f)
# list all left cosets with redundancies
local i,y,j,xH;
for j from 1 to nops(G) do
 xH:={};
 for i from 1 to nops(H) do
  y:=f(G[j],H[i]):
   xH:={op(xH),y}:
 od;
 print(xH);
od;
end:

LtoA:=proc(L)
 local i,A;
 A:=array(1..nops(L));
 for i from 1 to nops(L) do
  A[i]:=L[i]:
 od:
 RETURN(A);
end:

newf:=proc(a,b)
# uses CT(G,f) to make a new function that returns the same values as f
 local i,j,m,n,x:
 global CT,ans:
 x:=linalg[coldim](CT):
 m:=0:n:=0:
 for i from 1 to x while(m=0) do
  if(CT[1,i]=a) then m:=i: fi;
 od;
 for j from 1 to x while(n=0) do
  if(CT[1,j]=b) then n:=j: fi;
 od;
 if(m=0 or n=0) then RETURN(` a or b is not in the set`); fi;
 ans:=CT[m,n];
end:
```

```
Normalizer:=proc(H,G,f)
 #finds all x in G such that xHx^-1=H
 local id,H1,h,x,N;
 N:=[];
 id:=Identity(G,f);
 if id=noid then RETURN(failure) fi;
 for x in G do
  H1:={};
  for h in H do
   H1:={op(H1),f(f(x,h),Inverse(x,G,f))};
  od;
  if convert(H,set)=H1 then N:=[op(N),x] fi;
 od;
 RETURN(N);
end:

numbfntabel:=proc(n::integer)
 local i,tot,x;
 x:=op(2,ifactors(n));
 tot:=1:
 for i from 1 to nops(x) do
  tot:=tot*(combinat[numbpart](x[i][2])):
 od;
 RETURN(tot);
end:

partx:=proc(n::integer,a)
 local i,j,x,m;
# calls partition then takes a^(each element)
# used by fntabel
 x:=combinat[partition](n):
 m:=nops(x);
 x:=LtoA(x):
 for i from 1 to m do
  for j from 1 to nops(x[i]) do
   x[i][j]:=a^(x[i][j]);
  od;
 od;
 RETURN(x);
end:

pgp:=proc(n::integer,L::set)
 #L is a set of permutations of [1..n]
 #returns the elements of the group of degree
 #n generated by the elements of L
 local i,j,G,L0,g;
 global Pgp:
 L0:=combinat[permute](n);
 Pgp:=[];
 for i from 1 to nops(L0) do
  g[i]:=convert(L0[i],'disjcyc');
 od;
 G:=permgroup(n,L);
 for i from 1 to nops(L0) do
```

```
  if groupmember(g[i],G) then
   Pgp:=[op(Pgp),g[i]];
  fi;
 od;
 print(Pgp);
 print(`use mulperms`);
end:

q8:=proc()
 local i:
 global Q8,QT:
 QT:=matrix(8,8,[e,a,a^2,a^3,b,ba,ba^2,ba^3,a,a^2,a^3,e,
 ba^3,b,ba,ba^2,a^2,a^3,e,a,ba^2,ba^3,b,ba,a^3,e,a,a^2,
 ba,ba^2,ba^3,b,b,ba,ba^2,ba^3,a^2,a^3,e,a,ba,ba^2,ba^3,
 b,a,a^2,a^3,e,ba^2,ba^3,b,ba,e,a,a^2,a^3,ba^3,b,ba,ba^2,
 a^3,e,a,a^2]):
 Q8:=array(1..8):
 for i from 1 to 8 do
  Q8[i]:=QT[1,i]:
 od:
 Q8:=convert(Q8,list):
end:

qmult:=proc(a,b)
 local i,x,y:
 global ans:
 x:=0:y:=0:
 for i from 1 to 8 while(x=0) do
  if(Q8[i]=a) then x:=i: fi:
 od:
 for i from 1 to 8 while(y=0) do
  if(Q8[i]=b) then y:=i: fi:
 od:
 ans:=QT[x,y];
end:

Rcoset:=proc(x,H,G,f)
#find Hx
 local i,y,z,m,Hx;
 m:=member(x,G):
 if(m=false) then RETURN(` ERROR: not in group`);fi;
 Hx:={};
 for i from 1 to nops(H) do
  y:=f(H[i],x):
  z:=member(y,Hx):
  if (z=false) then
   Hx:={op(Hx),y}:
  fi;
 od;
 RETURN(Hx);
end:

sbgp:=proc(S,G,f)
#assumes S is in G and nonempty
#then uses the finite subgroup test to tell if S<G
```

```
 local i,j,x,y,ans:
 i:=Closed(S,f);
 if(i=closed) then RETURN(`Yes`); fi;
 RETURN(`No`);
end:

sn:=proc(n::integer)
 local i,j,G,L0,g;
 global Sn:
 L0:=combinat[permute](n);
 Sn:=[];
 for i from 1 to nops(L0) do
  g[i]:=convert(L0[i],'disjcyc');
 od;
 for i from 1 to nops(L0) do
  Sn:=[op(Sn),g[i]];
 od;
 print(`Sn=`,Sn);
 print(`use mulperms`);
end:

subgps:=proc(G,f)
# finds all the subgroups of G
# narrows down the possible orders by Lagrange's Theorem
# then choses all possible combinations of elements of G
# Any set without the identity is discarded. The remaining sets
# are tested with the finite subgroup test (closure).
 local c,i,j,k,DiV,X,y,id:
 DiV:=numtheory[divisors](nops(G));
 id:=Identity(G,f);
 for i from 1 to nops(DiV) do
  y:=[];
  X:=combinat[choose](G,DiV[i]);
  for j from 1 to nops(X) do
   k:=member(id,X[j]):
   if(k=true) then
     c:=Closed(X[j],f):
     if(c=closed) then
     y:=[op(y),X[j]]:
     fi;
   fi;
  od:
  print(y);
 od:
end:

subgpsord:=proc(n::integer,G,f)
#finds all subgroups of G of order n
#if n fails Lagrange's theorem, user is notified
#if n=1, the trivial subgroup is returned
#if n=|G|, then G is returned
#otherwise all combinations of n elements of G are tested
#for identity, then closure
 local y,c,j,k,X,Div,z,id:
 id:=Identity(G,f);
```

```
 Div:=numtheory[divisors](nops(G)):
 z:=member(n,Div):
 if(z=false) then RETURN(` No subgroups with that order by LaGrange`); fi;
 y:=[];
 if(n=1) then RETURN(id) fi;
 if(n=nops(G)) then RETURN(G); fi;
  X:=combinat[choose](G,n);
 for j from 1 to nops(X) do
  k:=member(id,X[j]):
  if(k=true) then
   c:=Closed(X[j],f):
   if(c=closed) then
    y:=[op(y),X[j]]:
   fi;
  fi;
 od:
 RETURN(y);
end:

t12:=proc()
 local i:
 global T12,TT:
 TT:=matrix(12,12,[1,A,B,C,D,E,F,G,H,I,J,K,A,B,C,D,E,1,G,H,I,J,K,
 F,B,C,D,E,1,A,H,I,J,K,F,G,C,D,E,1,A,B,I,J,K,F,G
 ,H,D,E,1,A,B,C,J,K,F,G,H,I,E,1,A,B,C,D,K,F,G,H
 ,I,J,F,K,J,I,H,G,C,B,A,1,E,D,G,F,K,J,I,H,D,C
 ,B,A,1,E,H,G,F,K,J,I,E,D,C,B,A,1,I,H,G,F,K,J,1
 ,E,D,C,B,A,J,I,H,G,F,K,A,1,E,D,C,B,K,J,I,H,G,F,B,A,1,E,D,C]):
 T12:=array(1..12):
 for i from 1 to 12 do
   T12[i]:=TT[1,i]:
 od:
 T12:=convert(T12,list):
end:

tmult:=proc(a,b)
 local i,x,y,ans:
 x:=0:y:=0:
 for i from 1 to 12 while(x=0) do
  if(T12[i]=a) then x:=i: fi:
 od:
 for i from 1 to 12 while(y=0) do
  if(T12[i]=b) then y:=i: fi:
 od:
 if(x=0 or y=0) then RETURN(` Error`); fi;
 ans:=TT[x,y];
end:

Umodn:=proc(n::integer,f)
 local i,j,H:
 global Un,ordun:
 H:={seq(i,i=0..n-1)}:
  Un:=[]:
   for j from 1 to n do:
   if(gcd(H[j],n)=1) then
```

```
    Un:=[op(Un),H[j]]:
   fi:
  od:
  printf(`Un = %a\n`,Un);
 f:=(a,b)->(a*b) mod n;
end:

Zmodn:=proc(n::integer,f)
 local i:
 global Zn:
 Zn:=[seq(i-1,i=1..n)]:
 printf(`Zn=%a\n`,Zn);
 f:=(a,b)-> (a+b) mod n;
end:
```

dihedral.mpl

```
Check:=proc(L::list)
 local i,x;
 if nops(L)<>11 then RETURN(error) fi;
 x:=0;
 for i to 10 do
  x:=md5(x,sig(i,L[i]));
 od;
 x:=md5(x,L[11]);
 if x=0 then RETURN(correct) fi;
 RETURN(`There is an error`);
end:

Check2:=proc(L)
local i,x,y;
if(nops(L)<>12) then RETURN(error) fi;
x:=0;y:=0;
for i to 5 do
 x:=md5(sigmap(i,L[2*i]),x);
 y:=md5(sigmap(i,L[2*i-1]),y);
od;
x:=md5(x,L[12]);
y:=md5(y,L[11]);
if x=0 and y=0 then RETURN(correct) fi;
RETURN(`there is an error`);
end:

checkdigits:=proc(L)
local i,x,y,m,n;
if nops(L)<>10 then RETURN(error) fi;
x:=0;y:=0;
for i to 5 do
 x:=md5(sigmap(i,L[2*i]),x);
 y:=md5(sigmap(i,L[2*i-1]),y);
od;
for i from 0 to 9 do
if md5(x,i)=0 then m:=i; break fi;
```

```
od;
for i from 0 to 9 do
if md5(y,i)=0 then n:=i; break  fi;
od;
RETURN([n,m]);
end:

D5:=matrix(10,10,[0,1,2,3,4,5,6,7,8,9,1,2,
3,4,0,6,7,8,9,5,2,3,4,0,1,7,8,9,5,6,3,4,0,1,
2,8,9,5,6,7,4,0,1,2,3,9,5,6,7,8,5,9,8,7,6,
0,4,3,2,1,6,5,9,8,7,1,0,4,3,2,7,6,5,9,8,2,1,
0,4,3,8,7,6,5,9,3,2,1,0,4,9,8,7,6,5,4,3,2,1,0]):

FormCheckDigit:=proc(L::list)
 local i,x;
 if nops(L)<>10 then RETURN(error) fi;
 x:=0;
 for i to 10 do
  x:=md5(x,sig(i,L[i]));
 od;
 for i from 0 to 9 do
  if md5(x,i)=0 then RETURN(i) fi;
 od;
end:

md5:=proc(m::integer,n::integer)
 if m>9 or m<0 then RETURN(ERROR) fi;
 if n>9 or n<0 then RETURN(error) fi;
 RETURN(D5[m+1,n+1]);
end:

sig:=proc(n::integer,a::integer)
 local i;
 if n<1 or n>10 then RETURN(error) fi;
 if a<0 or a>9 then RETURN(error) fi;
 RETURN(ST[n,a+1]);
end:

sigmap:=proc(p,x)
 local i,y;
 y:=x;
 for i to p do
 y:=sigma[y+1];
 od;
 RETURN(y);
end:

ST:=matrix(10,10,[1,5,7,6,2,8,3,0,9,4,5,8,0,3,7,
9,6,1,4,2,8,9,1,6,0,4,3,5,2,7,9,4,5,3,1,2,6,8,7,
0,4,2,8,6,5,7,3,9,0,1,2,7,9,3,8,0,6,4,1,5,7,0,4,
6,9,1,3,2,5,8,0,1,2,3,4,5,6,7,8,9,1,
5,7,6,2,8,3,0,9,4,5,8,0,3,7,9,6,1,4,2]):
```

RSA.mpl

```
nrs:=proc(p,q)
 local i,s,n,r,m;
 if not isprime(p) then RETURN(error) fi;
 if not isprime(q) then RETURN(error) fi;
 n:=p*q;
 m:=lcm(q-1,p-1);
 r:=nextprime(max(p,q));
 for i to m do
  if r*i mod m = 1 then s:=i; break fi;
 od;
 RETURN([n,r,s]);
end:

code:=proc(M,n,r)
 if gcd(n,M)<>1 then RETURN(Failure) fi;
 RETURN((M**r) mod n);
end:

decode:=proc(R,n,s)
 RETURN(R**s mod n);
end:

num:=proc(L)
 local i1,R,AN,W;
 AN:=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,_];
 R:=L;W:=table();
  for i1 to 27 do
   W[AN[i1]]:=i1;
  od;
 for i1 to nops(L) do
  R[i1]:=W[R[i1]];
 od;
 RETURN(R);
end:

alph:=proc(L)
 local i1,R,AN;
 AN:=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,_];
 R:=L;
  for i1 to nops(L) do
   R[i1]:=AN[R[i1]];
  od;
 RETURN(R);
end:

codemess:=proc(L,n,r)
 local i1,M;
 M:=num(L);
 for i1 to nops(M) do
 M[i1]:=code(M[i1],n,r);
 od;
 RETURN(M);
end:
```

```
decodemess:=proc(R,n,s)
 local i1,L;L:=R;
 for i1 to nops(L) do
 L[i1]:=decode(L[i1],n,s);
 od;
 RETURN(alph(L));
end:
```

AbStab.mpl

```
top:=[[1,3,8,6],[2,5,7,4],[9,33,25,17],[10,34,26,18],[11,35,27,19]]:
left:=[[9,11,16,14],[10,13,15,12],[1,17,41,40],[4,20,44,37],
[6,22,46,35]]:
front:=[[17,19,24,22],[18,21,23,20],[6,25,43,16],[7,28,42,13],
[8,30,41,11]]:
right:=[[25,27,32,30],[26,29,31,28],[3,38,43,19],[5,36,45,21],
[8,33,48,24]]:
rear:=[[33,35,40,38],[34,37,39,36],[3,9,46,32],[2,12,47,29],
[1,14,48,27]]:
bottom:=[[41,43,48,46],[42,45,47,44],[14,22,30,38],[15,23,31,39],
[16,24,32,40]]:

Gens:=[left,front,right,rear,bottom,top]:
print( ` left,front,right,rear,bottom,top`);

PG:=permgroup(48,{left,front,right,rear,bottom,top});

with(group):readlib(`group/stabchain`):readlib(`group/ordering`):
readlib(`group/tc`):

MakeStab:=proc(gens,b,res)
 local mp,base,gen,i,l,p,n,j,f,orb,trans,wantInverses;
 if gens=[] then RETURN([]) fi; if nargs=1 then base:=[];
 else base:=b;
  if not type(base,list) then base:=[];fi;
 fi;
 if nargs=3 then
  wantInverses:=evalb(res=true); else wantInverses:=false;
 fi;   n:=nops(gens[1]); if gens=[[`$`(1..n)]] then RETURN([]);fi;
 mp:=movedpoints(gens);
 for i to nops(mp) do
  if not member(mp[i],base) then base:=[op(base),mp[i]] fi;
 od;
 l:=1;f:=FORALL(gens,base[l]);
 while l<=nops(base) and f=true do
  l:=l+1;
 od;
 if l<= nops(base) then
  i:=base[l];
  orb:=[i];
  l:=1;
  trans:=table();
  trans[i]:=0;
  while l<=nops(orb) do
```

```
   i:=orb[l];
   for p to nops(gens) do
    gen:=gens[p];
    if wantInverses=true then
     j:=gen[i];
     if not member(j,orb) then
      orb:=[op(orb),j];
      trans[j]:=-p;
     fi;fi;
     j:=`group/ip`(gen)[i];
     if not member(j,orb) then
      orb:=[op(orb),j];
      trans[j]:=p;
     fi;
    od;
    l:=l+1;
   od;
  fi;
 [orb,trans]
end:

movedpoints:=proc(gens)
 local i,j,mp;
 mp:={};
 for i to nops(gens) do
  for j to nops(gens[i]) do
   if gens[i][j]<>j then mp:={op(mp),j} fi;
  od;
 od;
 RETURN(mp);
end:

FORALL:=proc(gens,bp)
 local g,ans;
 for g in gens do
  if g[bp]<>bp then RETURN(false) fi;
 od;
 RETURN(true);
end:

LeftSchreier:=proc(n,gens,agens,lsch)
 local c,i, asch, sch, x, a, t, G, g, ag;
 if nargs=3 then
  c:=MakeStab(gens,[],true);
  sch:=table();
  asch:=table();
  for x in c[1] do
   sch[x]:=[`$`(1..n)];
   asch[x]:=[];
   a:=x;
   while a<>c[1][1] do
    t:=c[2][a];
    if t>0 then
     g:=gens[t];
     ag:=agens[t];
```

```
    elif t<0 then
     g:=`group/ip`(gens[-t]);
    ag:=`group/invword`(agens[-t]);
    else
     g:=[`$`(1..n)];
     ag:=[];
    fi;
    sch[x]:=`group/mp`(sch[x],g);
    asch[x]:=`group/mulword`(asch[x],ag);
    a:=g[a];
   od;
  od;
 fi;
 [sch,asch,c[1]];
end:


Gentable:=proc(n,gens,agens,opt)
 local atable,Table,alt,art,orb,cnt, lt,rt,x,ag,g,at,t,i,j,k,aa,a,G;
 Table:=array(1..10000); cnt:=1;
 atable:=array(1..10000);
 if opt=true then lt:=LeftSchreier(n,gens,agens,opt);
 else lt:=LeftSchreier(n,gens,agens); fi;
 alt:=lt[2];orb:=lt[3];
 lt:=lt[1];
 rt:=[];art:=[];
 for i to n do  if type(lt[i],list) then
  rt:=[op(rt),`group/ip`(lt[i])];
  art:=[op(art),`group/invword`(alt[i])]; fi;
 od;
 for i to nops(gens) do
  for j to nops(rt) do
   at:=art[j];
   a:=`group/mp`(rt[j],gens[i]);
   aa:=`group/mulword`(at,agens[i]);
   k:=a[orb[1]];
   Table[cnt]:=`group/mp`(a,lt[k]);
   atable[cnt]:=`group/mulword`(aa,alt[k]);
   cnt:=cnt+1;
  od;
 od;
 [['Table[i]'$'i'=1..cnt-1],['atable[i]'$'i'=1..cnt-1]]
end:


MinGenSet:=proc(n,Table,Atable)
 local c,SC,i,m,j;
  c:=psort(Table,Atable);
 SC:=stabchain(n,c[1],c[2]); if SC<>[] then m:=nops(SC[1]);
 RETURN([['SC[1][i][1]'$'i'=1..m],SC[5]]); else RETURN([[],[]]);fi;
end:

psort:=proc(L1,L2)
 local x,i,R1,R2,T,n;
 n:=nops(L1);
```

```
 T:=array(1..n);R1:=array(1..n);R2:=array(1..n);
 for i to n do
  T[i]:=[L1[i],L2[i]];
 od;
 T:=sort(convert(T,list),(a,b)->evalb(nops(a[2])<nops(b[2])));
 for i to n do
  R1[i]:=T[i][1];
  R2[i]:=T[i][2];
 od;
 RETURN([['R1[i]'$'i'=1..n],['R2[i]'$'i'=1..n]]);
end:

FirstStabilizer:=proc(n,gens,agens,opt)
 local gt,c,H,x;
 gt:=Gentable(n,gens,agens,opt);
 H:=MinGenSet(n,gt[1],gt[2]);H:=psort(H[1],H[2]);
 RETURN(H);
end:

MakeAbStabChain:=proc(n,gen1,agens,base,x)
 local H,l,orb,gens,i,trans,M;
 if nargs=4 then l:=base;
 else l:=[];
 fi; gens:=gen1;if nargs=5 then for i to nops(gen1) do gens[i]:=convert(gens[i],plist,n); od; fi;
 M:=MakeStab(gens,l,false); if M=[] then RETURN([]);fi;
 orb:=M[1];
 trans:=M[2];
 if agens<>[] and agens<>[[]] then
  H:=FirstStabilizer(n,gens,agens,false);
  H:=MakeAbStabChain(n,H[1],H[2],l);
 fi;
 [gens,agens,orb,trans,H]
end:

WordModStabilizer:=proc(n,gens,agens,orb,trans,x)
 local abstract,a,s,g,ag,t;
 if orb=[] then RETURN([x,[]]) fi;
 abstract:=[];
 s:=x;
 a:=x[orb[1]];
 if not member(a,orb) then RETURN(NOT_IN_GROUP); fi;
 while a<>orb[1] do
  t:=trans[a];
  if t>0 then
   g:=gens[t];
  ag:=agens[t];
  elif t=0 then
    g:=[`$`(1..n)];
   ag:=[];
  else
   g:=`group/ip`(gen[-t]);
   ag:=`group/invword`(gen[-t]);
  fi;
  s:=`group/mp`(s,g);
  abstract:=`group/mulword`(abstract,ag);
```

```
 a:=g[a];
od;
RETURN([s,`group/invword`(abstract)]);
end:

Factorperm:=proc(n,elm,SC)
local elm1,abelm,l,S;
S:=SC;
elm1:=elm;
abelm:=[];
while elm1<>[] and elm1<>[`$`(1..n)] do
 l:=WordModStabilizer(n,S[1],S[2],S[3],S[4],elm1);
 elm1:=l[1];
 abelm:=`group/mulword`(l[2],abelm);
 if elm1<>[] and elm1<>[`$`(1..n)] then
  S:=S[5]; if S=[] then RETURN(NOT_IN_GROUP); fi;
 fi;
od;
RETURN(abelm);
end:

stabchain:=proc(n,gens,agens)
local  p, sc;
option remember, `Copyright (c) 1990 by the University of Waterloo. All rights reserved.`;
sc := [];
for p to nops(gens) do
 sc := addperm(gens[p], sc,agens[p])
od;
sc
end:

addperm:=proc(p, sc, q)
local i, n,agens, gens, fixed, creps, stab, newcreps, pl, pl2, orb, newcs;
option `Copyright (c) 1990 by the University of Waterloo. All rights reserved.`;
   n := nops(p);
   if `group/inchain`(p, sc) then RETURN(sc) fi;
   if sc = [] then
      gens := [[p, `group/ip`(p)]];
    if nargs=3 then
     agens:=[q];
    fi;
      fixed := -1;
      for i to n while fixed < 0 do
         if p[i] <> i then fixed := i fi
      od;
      creps := [[] $ n];
      creps := subsop(
         fixed = [[`$`(1 .. n)], [`$`(1 .. n)]], creps);
      stab := []
   else
      gens := [[p, `group/ip`(p)],op(sc[1])];
      if nargs=3 then agens:=[q,op(sc[5])]; fi;
      fixed := sc[2];
      creps := sc[3];
      stab := sc[4]
```

```
    fi;
    newcs := `group/findorb`(n, gens, fixed, creps);
    creps := newcs[2];
    newcreps := newcs[1];
    for pl in gens do for pl2 in newcreps do
          orb := pl[1][pl2[1][fixed]];
          stab := `group/addperm`(`group/mp`(pl2[1],
             `group/mp`(pl[1], creps[orb][2])), stab)
       od
    od;
    pl := gens[1];
    for i to n do
       pl2 := creps[i];
       if pl2 <> [] then
          orb := pl[1][pl2[1][fixed]];
          creps[orb];
          stab := `group/addperm`(`group/mp`(pl2[1],
             `group/mp`(pl[1], creps[orb][2])), stab)
       fi
    od;
   if nargs=3 then RETURN([gens,fixed,creps,stab,agens]); else
    [gens, fixed, creps, stab] fi;
end:

checker:=proc(x,G,AG,n)
local ans,i,k,c;
 ans:=['i'$'i'=1..n];
 for i to nops(x) do
  for k to nops(AG) do
  if x[i]=op(AG[k]) then ans:=`group/mp`(ans,G[k]); fi;
  if x[i]=1/op(AG[k]) then ans:=`group/mp`(ans,`group/ip`(G[k])); fi;
  od;
 od;
 RETURN(ans);
end:

Tremble:=proc(S1,S,n,elm,p)
 local i,w,x,z;
 w:=elm;
 for i to nops(S1) do
  x:=`group/mulword`(S1[i],elm);
  z:=checker(x,S[1],S[2],n);
  x:=`group/mulword`(`group/invword`(S1[i]),Factorperm(n,z,S));
 if nops(x)<nops(w) then
  w:=x;
 fi;
 od;
 RETURN(w);
end:

addinvs:=proc(S)
 local i,R;
 R:=S;
 for i to nops(S) do
  R:=[op(R),`group/invword`(S[i])];
```

```
 od;
 RETURN(R);
end:

ShortWords:=proc(R,pg,n,i1,i2,i3,i4)
 local i,j,X,Y,y,z;
 X:=R;
 Y:={};
  for i to nops(R) do
   Y:={op(Y),checker(R[i],pg[1],pg[2],n)};
  od;
  for i from i1 to i2 do
   for j from i3 to i4 do
    y:=`group/mulword`(R[i],R[j]);
    z:=checker(y,pg[1],pg[2],n);
    if not member(z,Y) then
     X:=[op(X),y]; Y:={op(Y),z};
    fi;
   od;
  od;
  RETURN(X);
end:

RCpermfind:=proc(L)
 local p,C,CS,i1,i2,i3;
 C[1]:=[[L[1],L[9],L[35]],[1,9,35]];
 C[2]:=[[L[3],L[27],L[33]],[3,27,33]];
 C[3]:=[[L[6],L[11],L[17]],[6,11,17]];
 C[4]:=[[L[8],L[19],L[25]],[8,19,25]];
 C[5]:=[[L[41],L[22],L[16]],[41,22,16]];
 C[6]:=[[L[43],L[24],L[30]],[43,24,30]];
 C[7]:=[[L[46],L[14],L[40]],[46,14,40]];
 C[8]:=[[L[48],L[32],L[38]],[48,32,38]];
 C[9]:=[[L[2],L[34]],[2,34]];
 C[10]:=[[L[4],L[10]],[4,10]];
 C[11]:=[[L[5],L[26]],[5,26]];
 C[12]:=[[L[7],L[18]],[7,18]];
 C[13]:=[[L[13],L[20]],[13,20]];
 C[14]:=[[L[21],L[28]],[21,28]];
 C[15]:=[[L[29],L[36]],[29,36]];
 C[16]:=[[L[37],L[12]],[37,12]];
 C[17]:=[[L[23],L[42]],[23,42]];
 C[18]:=[[L[31],L[45]],[31,45]];
 C[19]:=[[L[15],L[44]],[15,44]];
 C[20]:=[[L[39],L[47]],[39,47]];
 p:=[`$`(1..48)];
 CS[1]:=[[TP,LF,BK],[1,9,35]];
 CS[2]:=[[TP,RT,BK],[3,27,33]];
 CS[3]:=[[TP,LF,FT],[6,11,17]];
 CS[4]:=[[TP,FT,RT],[8,19,25]];
 CS[5]:=[[BT,FT,LF],[41,22,16]];
 CS[6]:=[[BT,FT,RT],[43,24,30]];
 CS[7]:=[[BT,LF,BK],[46,14,40]];
 CS[8]:=[[BT,RT,BK],[48,32,38]];
 CS[9]:=[[TP,BK],[2,34]];
```

```
 CS[10]:=[[TP,LF],[4,10]];
 CS[11]:=[[TP,RT],[5,26]];
 CS[12]:=[[TP,FT],[7,18]];
 CS[13]:=[[LF,FT],[13,20]];
 CS[14]:=[[FT,RT],[21,28]];
 CS[15]:=[[RT,BK],[29,36]];
 CS[16]:=[[BK,LF],[37,12]];
 CS[17]:=[[FT,BT],[23,42]];
 CS[18]:=[[RT,BT],[31,45]];
 CS[19]:=[[LF,BT],[15,44]];
 CS[20]:=[[BK,BT],[39,47]];
 for i1 to 8 do
  for i2 to 8 do
   if convert(C[i1][1],set)=convert(CS[i2][1],set) then
   for i3 to 3 do
     if C[i1][1][i3]=CS[i2][1][1] then
      p[C[i1][2][i3]]:=CS[i2][2][1];
     elif C[i1][1][i3]=CS[i2][1][2] then
      p[C[i1][2][i3]]:=CS[i2][2][2];
     elif C[i1][1][i3]=CS[i2][1][3] then
       p[C[i1][2][i3]]:=CS[i2][2][3];
     fi;
    od;
   break;
   fi;
  od;
 od;
 for i1 from 9 to 20 do
  for i2 from 9 to 20 do
   if convert(C[i1][1],set)=convert(CS[i2][1],set) then
    for i3 to 2 do
     if C[i1][1][i3]=CS[i2][1][1] then
      p[C[i1][2][i3]]:=CS[i2][2][1];
     elif C[i1][1][i3]=CS[i2][1][2] then
       p[C[i1][2][i3]]:=CS[i2][2][2];
     fi;
    od;
    break;
   fi;
  od;
 od;
 RETURN(p);
end:
```

INDEX OF PROCEDURES MENTIONED IN THE TEXT

VITA

Thomas E. Cooper, III, was born in Knoxville, TN on September 13, 1978. His family moved to Dandridge when he was in the third grade. He graduated as valedictorian of Jefferson County High School in 1996. From there, he attended the University of Tennessee at Knoxville, where he graduated as the Top Graduate in the College of Arts and Sciences in August of 2000, receiving a B.S. degree in mathematics. He will receive a M.S. in mathematics in May of 2001.